UNITED STATES PATENT AND TRADEMARK OFFICE

BEFORE THE PATENT TRIAL AND APPEAL BOARD

Google LLC

Petitioner

v.

UNILOC 2017 LLC

Patent Owner

U.S. Patent No. 8,407,609
Filing Date: August 21, 2009
Issue Date: March 26, 2013

Case No. IPR2020-00115

DECLARATION OF DR. JEFFREY CHASE, Ph.D. IN SUPPORT OF PETITION FOR *INTER PARTES* REVIEW OF UNITED STATES PATENT NO. 8,407,609

TABLE OF CONTENTS

I, Dr. Jeffrey Chase, Ph.D., declare as follows:

I. Introduction

1. My name is Dr. Jeffrey Chase. I have been asked to submit this declaration on behalf of Google LLC (“Google” or “Petitioner”) in connection with a petition for *inter partes* review of U.S. Patent No. 8,407,609 (“the ’609 patent”), which I understand is being submitted to the Patent Trial and Appeal Board of the United States Patent and Trademark Office by Google.

2. I have been retained as a technical expert by Google to study and provide my opinions on the technology claimed in, and the patentability or non-patentability of, claims 1–3 of the ’609 patent (“the Challenged Claims”).

3. This declaration is directed to the Challenged Claims of the ’609 patent and sets forth certain opinions I have formed, the conclusions I have reached, and the bases for each.

4. Based on my experience, knowledge of the art at the relevant time, analysis of prior art references, and the understanding a person of ordinary skill in the art would have of the claim terms, it is my opinion that each of the Challenged Claims of the ’609 patent is unpatentable over the prior art references discussed below.

II. Background and Qualifications

5. I am a Professor at Duke University in the Computer Science Department. I have studied and practiced in the field of computer science for over 35 years. During this time, I have worked as a software developer, computer systems researcher, and computer science professor. I have been teaching Computer Science at Duke since 1995.

6. I received my Doctor of Philosophy (Ph.D.) degree in the field of Computer Science from the University of Washington in Seattle in 1995. I received my Masters of Science (M.S.) degree in Computer Science from the University of Washington in 1989. As a graduate student at the University of Washington, I conducted research on new operating system models for secure data sharing. I earned my Bachelor of Arts (B.A.) degree as a double major in Mathematics and Computer Science from Dartmouth College.

7. From 1985 through 1994 (before and during graduate school), I worked as a software design engineer at Digital Equipment Corporation ("DEC"), earning the title Senior Software Engineer in 1987. While at DEC, I developed operating system kernel software for networked file services in DEC's Unix operating system product, Ultrix.

8. Upon receiving my Ph.D. degree, I joined the faculty of Duke University in the Department of Computer Science as an Assistant Professor.

Since becoming a professor, I have conceived and led a number of research projects and published widely in leading research forums in the areas of operating systems and network services including high-performance Web systems and cloud computing. I earned tenure at Duke University in 2002, and was promoted to Full Professor in 2006. I teach courses for undergraduate and graduate students at Duke on various related subjects: operating systems, networking and networked systems, distributed systems, and Internet technology and society. I have supervised the research of fourteen completed Ph.D. dissertations in the field of Computer Science. I have also supervised the research of twenty students who earned Master's degrees at Duke.

9. My work has focused on software systems for efficient, secure, and reliable sharing of resources and information in computer networks ranging from clusters (e.g., cloud computing services) to the global Internet. I have conducted research and developed software relating to networked data sharing including cloud computing and high-performance Web systems and storage. I am a named inventor on eleven U.S. patents and a co-author of over 100 published research papers on related topics in peer-reviewed technical publications or conferences in the field of Computer Science.

10. I have served on editorial program committees for leading annual academic conferences in networked computer systems, cloud computing, storage,

Web technologies, and related areas. For example, I was invited to serve on the editorial program committee for the Association for Computing Machinery (ACM) Symposium on Cloud Computing (SoCC) multiple times (most recently in 2019) and co-chaired the SoCC committee in 2011. SoCC and other related venues are sponsored by the ACM, a leading professional society, of which I am a lifetime member. I have had similar roles in other related academic venues.

11. I conducted research in various Web technologies early in the Web computing era (mid-1990s) and up until the time the provisional application leading to the '609 patent was filed (2008). I have taught certain Web technologies in my courses, including Web service technologies based on the Java programming language, and I developed Java-based Web application software as part of my research (e.g., the Web interface for Shirako, an early cloud computing system, in 2005-2007). Much of my research during this period focused on technologies for high-performance Web services and led into my later research on cloud computing.

12. I have also participated in a number of industry collaborations. I am a named co-inventor of patents relating to Web caching and resource management in Web services resulting from these collaborations. While a collaborator at AT&T Corporation in 1996, I developed early technology for Web caching, patented as U.S. Patent No. 5,944,780 entitled "Network with Shared Caching." In collaboration with IBM Corporation from 2000–2003, I developed technology

covered by seven patents relating to adaptive resource management and request routing for hosted Web services.

13. In the course of my research, I have gained exposure to client-side Web technologies used to build these Web services. For example, the '609 patent describes Java applet technology and its use to add programmatic functions—such as tracking—that run in a user computer's browser as it displays a Web page. When the Java applet technology was first coming into use (around 1996–1998), I collaborated with IBM Corporation to develop a tool that could "instrument" or inject new code elements directly into compiled Java "bytecode" as it loads into a browser or other process. This collaboration was described in, for example:

- G. Cohen, J. Chase & D. Kaminsky, *Automatic Program Transformation with JOIE*, USENIX TECHNICAL CONFERENCE (June 1998); and
- G. Cohen & J. Chase, *An Architecture for Safe Bytecode Insertion*, available at https://www2.cs.duke.edu/ari/joie/.

14. Additional details about my employment history, fields of expertise, awards, publications, and other activities are further included in my curriculum vitae (which I have been told is Ex. 1004 to Google's petition).

15. I am being compensated for services provided in this matter at my customary rate, plus travel expenses. My compensation is not conditioned on the

conclusions I reach as a result of my analysis or on the outcome of this matter. Similarly, my compensation is not dependent upon and in no way affects the substance of my statements in this declaration.

16. I have no financial interest in Petitioner or any of its subsidiaries. I also do not have any financial interest in Patent Owner Uniloc 2017 LLC. I do not have any financial interest in the ’609 patent and have not had any contact with the named inventor of the ’609 patent (Tod C. Turner).

III. Materials Reviewed

17. In forming my opinions regarding the ’609 patent, I reviewed the following materials:

- The ’609 patent (which I have been told is Ex. 1001 to Google’s petition);
- U.S. Patent App. Pub. No. 2004/0045040 to Hayward (“*Hayward*,” which I have been told is Ex. 1005 to Google’s petition);
- U.S. Patent App. Pub. No. 2002/0111865 to Middleton (“*Middleton*,” which I have been told is Ex. 1006 to Google’s petition);
- U.S. Patent No. 6,421,675 to Ryan (“*Ryan*,” which I have been told is Ex. 1007 to Google’s petition);

- Defendant Google LLC's Claim Term Disclosure in *Uniloc 2017 LLC v. Google LLC*, No. 2:18-cv-00502 (E.D. Tex. Sep. 24, 2019) (which I have been told is Ex. 1008 to Google's petition);
- Plaintiffs' Preliminary Claim Constructions and Identification of Extrinsic Evidence Pursuant to P.R. 4-2 in *Uniloc 2017 LLC v. Google LLC*, No. 2:18-cv-00502 (E.D. Tex. Sep. 24, 2019) (which I have been told is Ex. 1009 to Google's petition);
- DAVID FLANAGAN, JAVASCRIPT: THE DEFINITIVE GUIDE 255 (5th ed. 2006) (attached as Appendix A); and
- CLARK S. LINDSEY ET AL., JAVATECH (2005) (attached as Appendix B);
- Aleksander Malinowski & Bogdan Wilamowski, *Internet Technology as a Tool for Solving Engineering Problems*, PROCEEDINGS OF IECON'01: THE 27TH ANNUAL CONFERENCE OF THE IEEE INDUSTRIAL ELECTRONICS SOCIETY 1622 (2001) (attached as Appendix C).

IV. Legal Standards

18. I am not an attorney and have not been asked to offer my opinion on the law. However, as an expert offering an opinion on whether the claims in the '609 patent are patentable, I have been told that I am obliged to follow existing law.

A. Anticipation

19. I have been told the following legal principles apply to analysis of patentability pursuant to 35 U.S.C. § 102, a provision in the patent law regarding anticipation. I have been told that, in an *inter partes* review proceeding, patent claims may be deemed unpatentable if it is shown by preponderance of the evidence that they were anticipated by one or more prior art patents or publications.

20. I have been told that for a claim to be anticipated under § 102, every limitation of the claimed invention must be disclosed by a single prior art reference, viewed from the perspective of a person of ordinary skill in the art.

21. I have been told that a claim is unpatentable as anticipated under § 102(b) if the claimed invention was "patented or described in a printed publication in this or a foreign country or in public use or on sale in this country, more than one year prior to the date of the application for patent in the United States."

22. I have been told that a claim is unpatentable as anticipated under § 102(e) if "the invention was described in (1) an application for patent, published under section 122(b), by another filed in the United States before the invention by the applicant for patent or (2) a patent granted on an application for patent by another filed in the United States before the invention by the applicant for patent,

except that an international application filed under the treaty defined in section 351(a) shall have the effects for the purposes of this subsection of an application filed in the United States only if the international application designated the United States and was published under Article 21(2) of such treaty in the English language."

B. Obviousness

23. I have been told the following legal principles apply to analysis of patentability pursuant to 35 U.S.C. § 103(a), a provision in the patent law regarding obviousness that reads "[a] patent may not be obtained although the invention is not identically disclosed or described as set forth in section 102, if the differences between the subject matter sought to be patented and the prior art are such that the subject matter would have been obvious at the time the invention was made to a person having ordinary skill in the art to which said subject matter pertains." I have been told that, in an *inter partes* review proceeding, patent claims may be deemed unpatentable if it is shown by a preponderance of the evidence that they were rendered obvious by one or more prior art patents or publications.

24. When considering the issues of obviousness, I have been told that I am to do the following:

a. Determine the scope and content of the prior art;

b. Ascertain the differences between the prior art and the claims at issue;

c. Resolve the level of ordinary skill in the pertinent art; and

d. Consider evidence of secondary indicia of non-obviousness (if available).

25. I have been told that the relevant time for considering whether a claim would have been obvious to a person of ordinary skill in the art is the time of alleged invention, which I have assumed is shortly before the provisional application leading to the ’609 patent was filed.

26. I have been told that obviousness is a determination of law based on underlying determinations of fact. I have been told that these factual determinations include the scope and content of the prior art, the level of ordinary skill in the art, the differences between the claimed invention and the prior art, and secondary considerations of non-obviousness.

27. I have been told that any assertion of secondary indicia must be accompanied by a nexus between the merits of the invention and the evidence offered.

28. I have been told that a reference may be combined with other references to disclose each element of the invention under § 103. I have been told that a reference may also be combined with the knowledge of a person of ordinary

skill in the art and that this knowledge may be used to combine multiple references. I have also been told that a person of ordinary skill in the art is presumed to know the relevant prior art. I have been told that the obviousness analysis may account for the inferences and creative steps that a person of ordinary skill in the art would employ.

29. In determining whether a prior art reference could have been combined with another prior art reference or other information known to a person having ordinary skill in the art, I have been told that the following principles may be considered:

a. A combination of familiar elements according to known methods is likely to be obvious if it yields predictable results;
b. The substitution of one known element for another is likely to be obvious if it yields predictable results;
c. The use of a known technique to improve similar items or methods in the same way is likely to be obvious if it yields predictable results;
d. The application of a known technique to a prior art reference that is ready for improvement, to yield predictable results;
e. Any need or problem known in the field and addressed by the reference can provide a reason for combining the elements in the manner claimed;

f. A person of ordinary skill often will be able to fit the teachings of multiple references together like a puzzle; and

g. The proper analysis of obviousness requires a determination of whether a person of ordinary skill in the art would have a "reasonable expectation of success"—not "absolute predictability" of success—in achieving the claimed invention by combining prior art references.

30. I have been told that whether a prior art reference renders a patent claim unpatentable as obvious is determined from the perspective of a person of ordinary skill in the art. I have been told that there is no requirement that the prior art contain an express suggestion to combine known elements to achieve the claimed invention, but a suggestion to combine known elements to achieve the claimed invention may come from the prior art, as filtered through the knowledge of one skilled in the art. In addition, I have been told that the inferences and creative steps a person of ordinary skill in the art would employ are also relevant to the determination of obviousness.

31. I have been told that, when a work is available in one field, design alternatives and other market forces can prompt variations of it, either in the same field or in another. I have been told that if a person of ordinary skill in the art can implement a predictable variation and would see the benefit of doing so, that variation is likely to be obvious. I have been told that, in many fields, there may

be little discussion of obvious combinations, and in these fields market demand—not scientific literature—may drive design trends. I have been told that, when there is a design need or market pressure and there are a finite number of predictable solutions, a person of ordinary skill in the art has good reason to pursue those known options.

32. I have been told that there is no rigid rule that a reference or combination of references must contain a "teaching, suggestion, or motivation" to combine references. But I also understand that the "teaching, suggestion, or motivation" test can be a useful guide in establishing a rationale for combining elements of the prior art. I have been told that this test poses the question as to whether there is an express or implied teaching, suggestion, or motivation to combine prior art elements in a way that realizes the claimed invention, and that it seeks to counter impermissible hindsight analysis.

V. Level of Ordinary Skill in the Art

33. I have been asked to provide a definition for the level or ordinary skill in the art. I have been informed that several factors are considered in assessing the level of ordinary skill in the art, including: (1) the types of problems encountered in the art; (2) the prior art solutions to those problems; (3) the rapidity with which innovations are made; (4) the sophistication of the technology; and (5) the educational level of active workers in the field. Based on my experience and

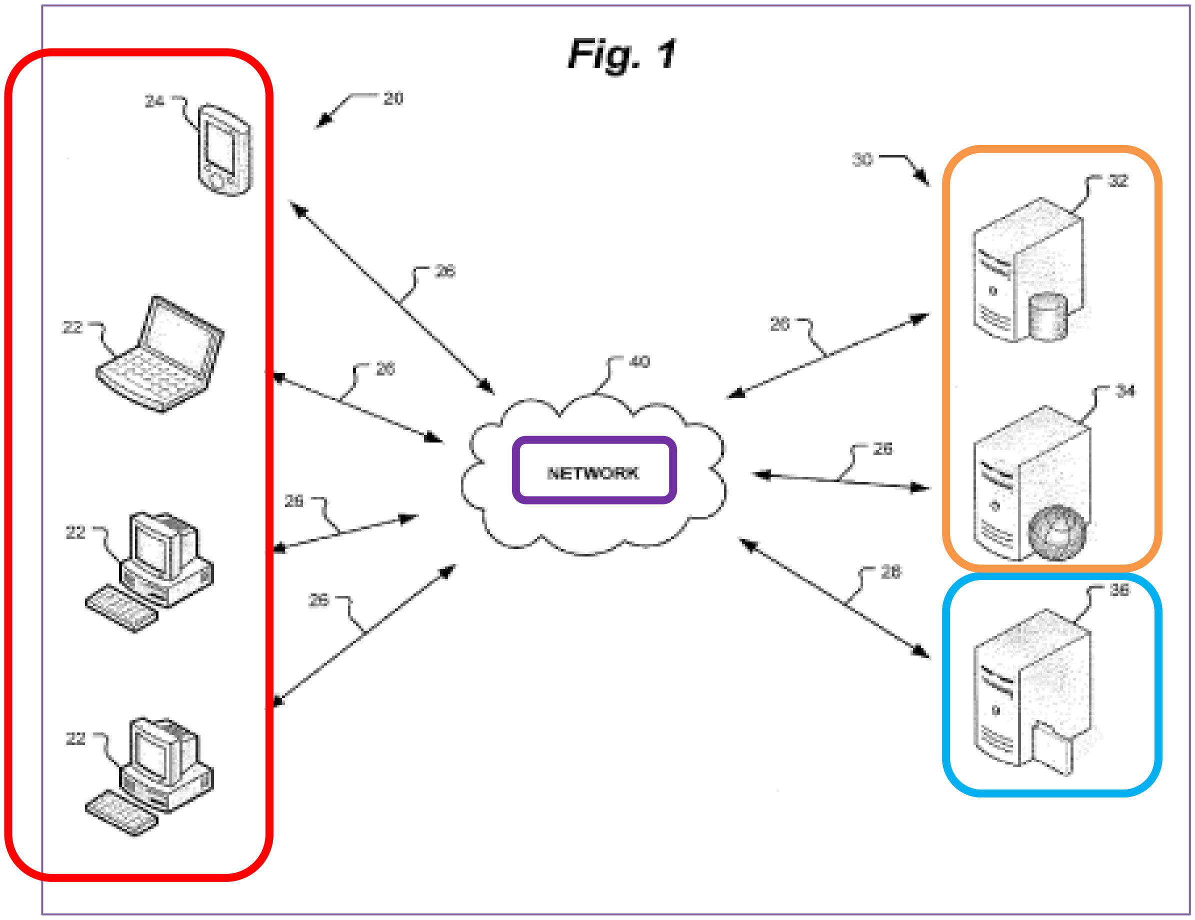
considering these factors, my opinion is that a person of ordinary skill in the art ("POSA") at the time of the filing of the provisional application leading to the '609 patent would have had either (a) a Master's or doctoral degree in computer science, electrical engineering, or a similar discipline involving relevant experience; or (b) a Bachelor's degree in computer science, electrical engineering, or a similar discipline and at least two years additional relevant experience. Working in the design and implementation of networked computing systems constitutes relevant work experience. Examples of such work in networked computing systems could include work in networked computing communication and data streaming.

34. I have not analyzed the priority date of the '609 patent, but I note that the earliest claim of priority listed on the face of the '609 patent is August 21, 2008, the filing date of U.S. Provisional Patent Application No. 61/090,672. Because all of the prior art discussed in this declaration pre-dates August 21, 2008, I have assumed for simplicity that August 21, 2008, is the priority date for the '609 patent. I have therefore also treated this date as the date from which to assess the knowledge available to a person of ordinary skill in the art. I note that I was at least a person or ordinary skill in the art as of this date.

VI. The ’609 Patent

A. Overview of the ’609 Patent

35. The ’609 patent describes a method for tracking digital media presentations delivered to a user’s computer. ’609 patent, Abstract. The method is carried out by the system shown in annotated Figure 1, below, which includes a user computer 20, a content or web server 34 and database server 32, and a file server 36, all of which are connected by a network 40.

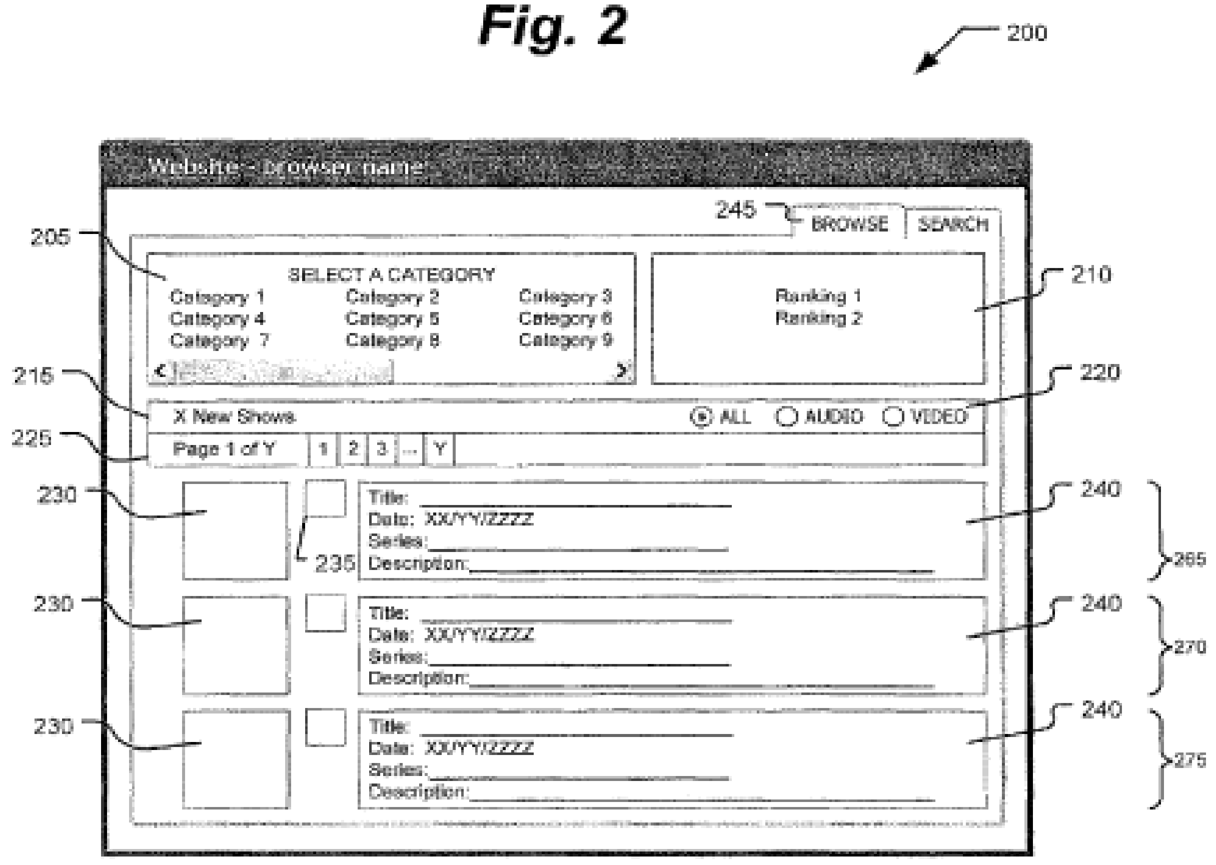


36. The ’609 patent teaches that “a user of a device 20 may request [a web] page 200 from content server 34 using a browser application,” “[s]erver 34

may provide page 200 to the requesting computer 20," and "[a] user may enter a search term." *Id.* at 4:57–61, 5:29–34. "Responsively thereto," the '609 patent explains, "content server 34 may request database server 32 to identify which presentations should be used to populate page 200 according to the entered search term(s)." *Id.* at 5:34–37.

37. "Server 34 may then provide such a populated page 200 to the requesting user computer 20." *Id.* at 5:37–39. An example web page 200 showing "aggregate[d] . . . video content for presentation to users of computers 20" is shown in Figure 2, below.

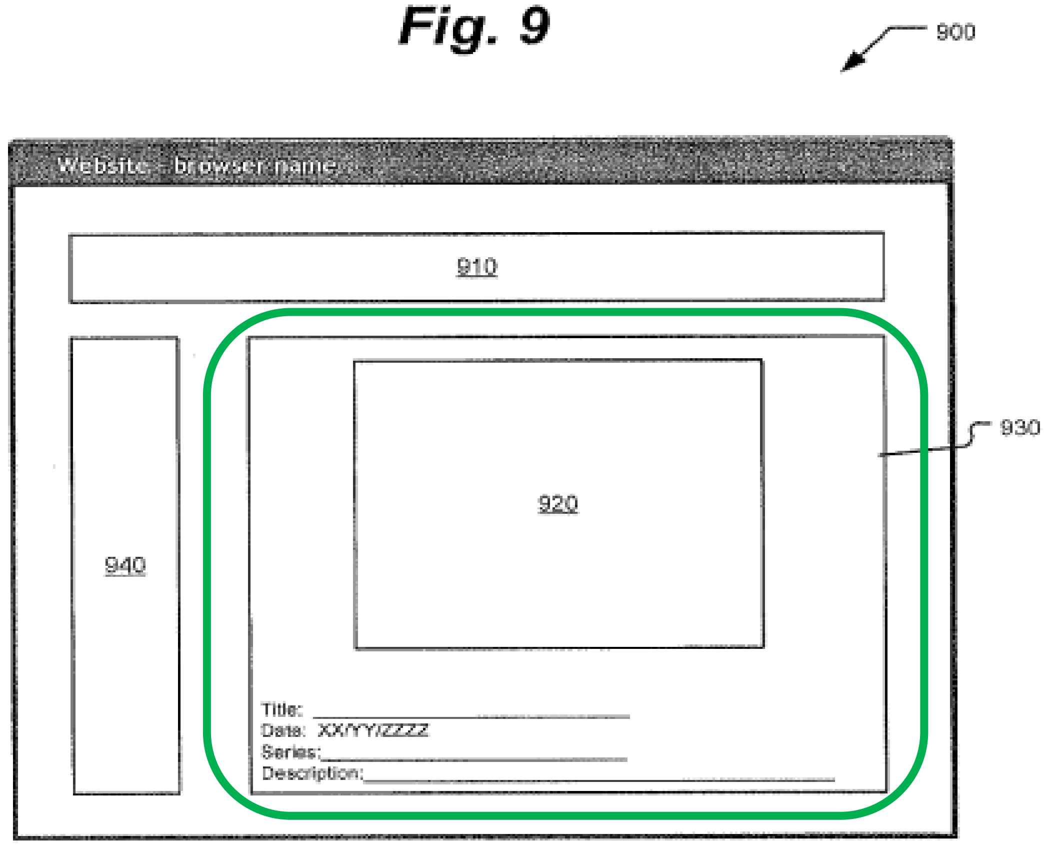
Fig. 2



38. As Figure 2 illustrates, “presentations 265, 270, 275” may be shown. *Id.* at 4:38–40. Then, “a user may select a populated presentation (e.g., 265, 270, or 275, FIG. 2)” and, “[i]n response thereto, server 34 may request file server 36 . . . stream . . . the selected presentation to the requesting user’s computer 20, such as via web page 200 in a conventional manner.” *Id.* at 5:20–25.

39. A “[w]eb page 900,” shown in Figure 9 below, “may be provided to user’s computer [20] responsively to user selection of a presentation shown on a populated web page 200.” *Id.* at 11:61–64. On the web page 900, a “portion 930 [green] may be utilized to playback the selected presentation in a conventional

Fig. 9



manner, e.g., by . . . streaming the content to a media player application or plug-in." *Id.* at 12:1–5.

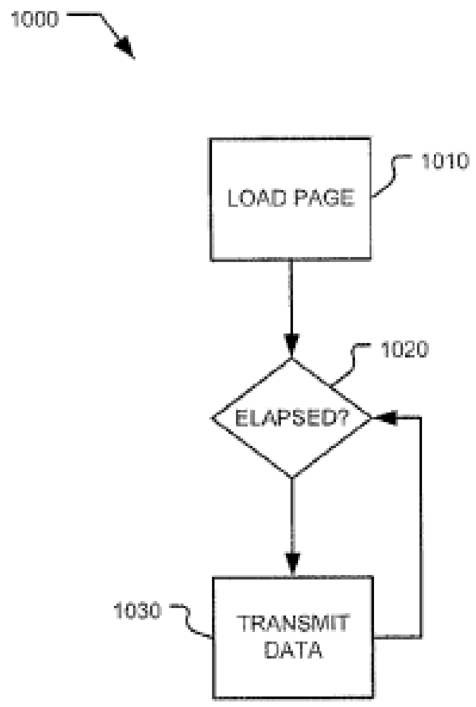
40. The '609 patent states "it may be desirable to know . . . how long a user actually watched, and/or listened, to a presented program." *Id.* at 11:47–52. For example, where advertisements are displayed in the web page alongside the presentation, "it may be desirable to be able to reliable [sic] identify how long the media was actually . . . played, in order to appropriately value portions [of the web page] as available advertising billboard space." *Id.* at 12:5–10.

41. But while *Hayward* discloses tracking how a user views a media file, including how long the media file was played, the '609 patent claims "[s]uch knowledge is not conventionally available." *Id.* at 13:47–48. The '609 patent tracks how the user views the digital media presentation using a "timer applet." *Id.* at 12:66–67.

42. As shown in Figure 10, below right, the timer applet "may be used to indicate each time some temporal time period, such as 10, 15, or 30 seconds, elapses." *Id.* at 13:6–9. "[W]hen the applet determines the predetermined temporal period has elapsed, . . . system 30 may log receipt of this indication, such

as by using database server 32." *Id.* at 13:10–13. In some embodiments, the applet may cause "identifying data" to be transmitted with the indication. *Id.* at 13:14–16. The identifying data may be "logged, such as by using database server 32." *Id.* at 13:22–23.

Fig. 10



43. Based on the logged data, it may be determined "that a viewer began viewing a particular show at a certain time," as well as "when a user began viewing a different page, or show, thereby providing knowledge of how long a particular viewer spent on a particular page." *Id.* at 13:43–48.

44. The '609 patent recognizes the value of this information to advertisers. Using this information, the '609 patent envisions, "an increasing scale of payments for advertising displayed on a given page" could be determined, "correspondent to how long a viewer or viewers remain, or typically remain, on that particular page." *Id.* at 13:49–14:2.

B. Challenged Claims

45. Google challenges claims 1–3.

46. For convenience, the Challenged Claims are reproduced below. I have added reference numerals for ease of reference:

Claim	Claim Language
1	1[a]. A method for tracking digital media presentations delivered from a first computer system to a user’s computer via a network comprising: [1b]. providing a corresponding web page to the user’s computer for each digital media presentation to be delivered using the first computer system; [1c]. providing identifier data to the user’s computer using the first computer system; [1d]. providing an applet to the user’s computer for each digital media presentation to be delivered using the first computer system, wherein the applet is operative by the user’s computer as a timer; [1e]. receiving at least a portion of the identifier data from the user’s computer responsively to the timer applet each time a predetermined temporal period elapses using the first computer system; and [1f]. storing data indicative of the received at least portion of the identifier data using the first computer system;

Claim	Claim Language
	[1g]. wherein each provided webpage causes corresponding digital media presentation data to be streamed from a second computer system distinct from the first computer system directly to the user's computer independent of the first computer system; [1h]. wherein the stored data is indicative of an amount of time the digital media presentation data is streamed from the second computer system to the user's computer; and [1i]. wherein each stored data is together indicative of a cumulative time the corresponding web page was displayed by the user's computer.
2	[2] The method of claim 1, wherein the storing comprises incrementing a stored value dependently on the receiving.
3	[3] The method of claim 2, wherein the received data is indicative of a temporal cycle passing.

C. Claim Construction

47. I have been told that claim terms in *inter partes* review proceedings are to be given their ordinary and accustomed meaning as understood by a person of ordinary skill in the art. In my analysis of the Challenged Claims, I have therefore applied the plain and ordinary meaning as understood by a person of ordinary skill in the art. I understand that Petitioner and Patent Owner have

offered various constructions in related litigation. Ex. 1008; Ex. 1009. For purposes of this proceeding, it is my opinion that the Board does not need to expressly construe the claims because it is my opinion that the Challenged Claims are unpatentable under both parties' constructions.

VII. Overview of Prior Art References

A. *Hayward*

48. I note that *Hayward* was published on March 4, 2004, which was more than one year before August 21, 2008. I am told that makes *Hayward* prior art to the '609 patent under § 102(b).

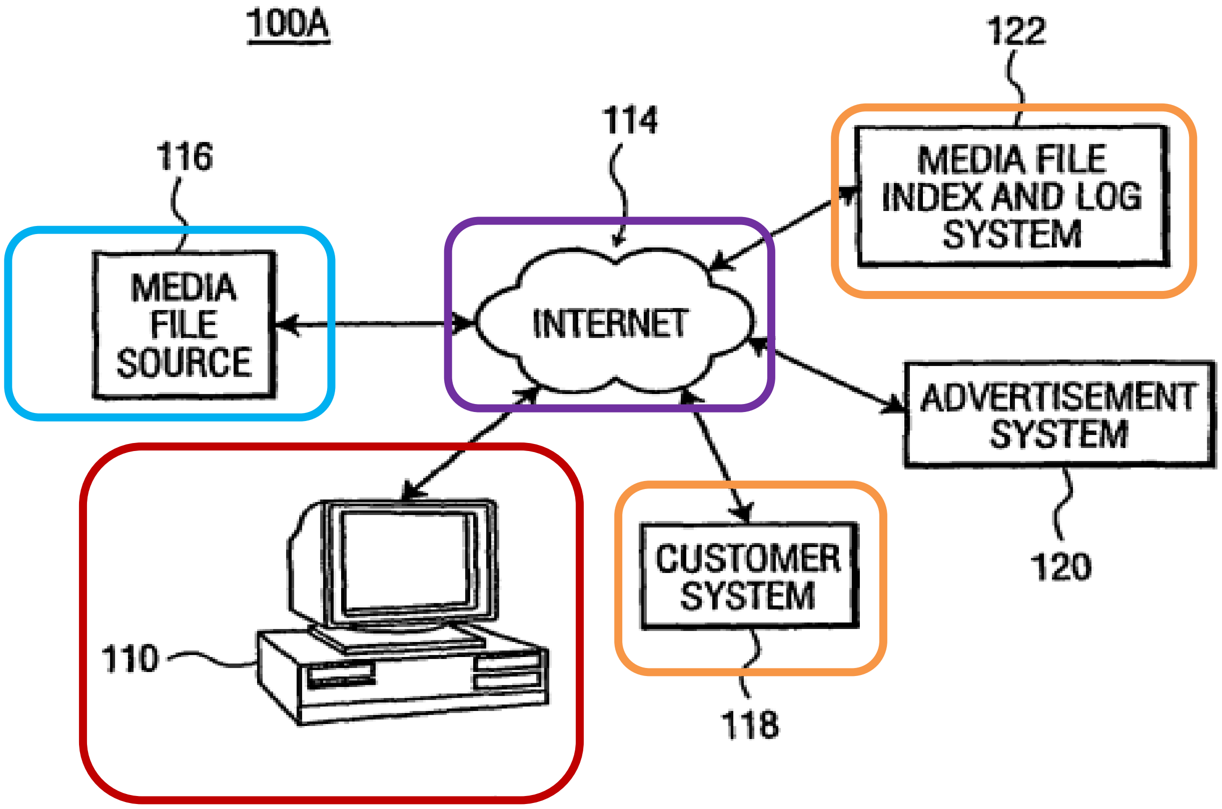


FIG. 1A

49. *Hayward* teaches "a method of displaying video data using an embedded media player page." *Hayward*, Abstract. As shown in annotated Figure 1A above, *Hayward*'s method is carried out by a system including a client 110, a customer system 118 and media file index and log system 122, and a media file source 116, all of which are connected by the Internet 114. While customer system 118 and media file index and log system 122 are shown separately, *Hayward* explains that these systems may be "combined physically within one . . system[]" in some embodiments. *Id.* ¶0030.

50. In *Hayward*, "[a] user of client 110 accesses customer system 118 through Internet 114," and "customer system 118 transmits a web page to client 110 through Internet 114." *Id.* ¶0025. The transmitted web page includes "a media file search prompt" where the user may enter a search request, such as "Pearl Harbor" and "movie trailer." *Id.* ¶¶0026, 0028. "The search request is received by customer system 118 and is transmitted to media file index and log system 122 . . . ." *Id.* ¶0028.

51. The media file index and log system 122 "includes a database having indexed therein a plurality of media files," each of which is identified by "a unique identifier for the media file." *Id.* ¶0027. When the search request is transmitted by the customer system 118, the media file index and log system 122 searches "for indexed media files that satisfy the search request" and "transmits the results to

customer system 118." *Id.* ¶0028. The search results include "the playing length of each video file, the URI address of each video file, encoding bit rate of the video file, file format, a database identifier unique to each video file, frame dimensional data for each video file, or any other information contained within the database." *Id.*

52. The customer system 118 transmits the search results to client 110 "as a Web page that preferably includes a list of links to media files located at media file sources 116." *Id.* The client 110 displays the web page with the search results to the user. *Id.*

53. The user may "view the video data contained within a video file listed in the search results displayed to the user by clicking a link to one of the video files." *Id.* ¶0029. When the user clicks a link for a selected media file, "the customer system 122 instructs the client to request [an] embedded media player page from the customer system 122." *Id.*

54. Figure 2, annotated below, shows an embedded media player page 200.

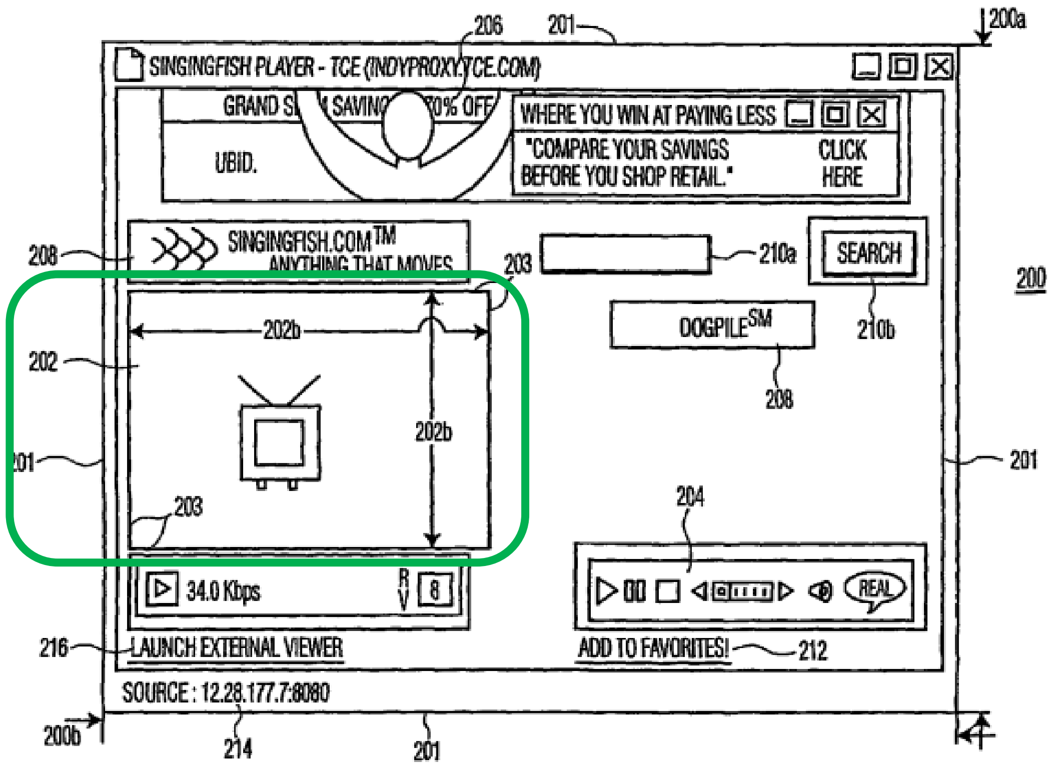


FIG. 2

55. The embedded media player page 200 "includes a reference to a functional media player object" that can play the selected media file. *Id.* ¶0031. The media player "is an application that outputs audio and video files that are either stored locally in a multimedia device, or are streamed/downloaded from a remote storage site via a communications interface." *Id.* ¶0002. The media player is "embedded" because it is "viewed . . . within a data display," like a web page. *Id.* ¶0002. For example, in Figure 2, a "video display area 202" is included within the web page 200. *Id.* ¶0032.

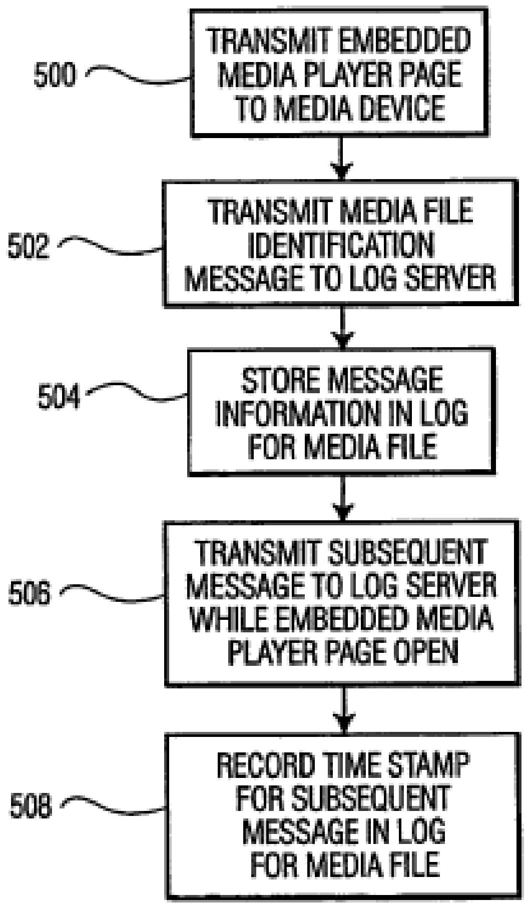
56. In *Hayward*, the "display code" for the web page may include "scripting that calls [the] media player, resident on the client, as an object for

embedding within the data display." *Id.* ¶0002. *Hayward* defines "scripting" as "server or client-side programming which supplements a static HTML page." *Id.* ¶0017. *Hayward* contemplates that the "scripting" could be, for example, "Java" or "JavaScript." *Id.*

57. In *Hayward*, the embedded media player page may locate a media file at the media file source 116 and stream the media file, which may be output by the media player. *Id.* ¶0046. For example, a media file titled "thestream.asx" may be "found and streamed by the media player from a media file source 116 located at 'thestreamhost.com' through Internet 114." *Id.* In particular, HTML code of the embedded media player page may find and stream the media file using a source command, such as "SRC=http://thestreamhost.com/thestream.asx." *Id.* ¶¶0044, 0046. The streamed media file may then be "outputted by the embedded media player in a window 202," as shown in annotated Figure 2 above.

58. In some embodiments, *Hayward* teaches, the "embedded media player page also facilitates the collection of data in connection with the playing of a media file." *Id.* ¶0057. *Hayward*'s data collection is described in connection with Figure 5, below right. As shown, "[a]t step 502, the embedded media player page instructs the client 110 to transmit a media file identification message to a log

server of the media file index and log system 122." *Id.* ¶0058. The media file identification message is transmitted "substantially proximate in time to when the media file begins to play in the embedded media player of the embedded media player page." *Id.* ¶0059. The media file identification message may include "the Internet Protocol (IP) address of the user, . . . the IP address or Universal Resource Locator of client 110, the domain address of the customer system 118 that transmitted the embedded media player page at the client 110, a unique identifier to



500
TRANSMIT EMBEDDED MEDIA PLAYER PAGE TO MEDIA DEVICE
502
TRANSMIT MEDIA FILE IDENTIFICATION MESSAGE TO LOG SERVER
504
STORE MESSAGE INFORMATION IN LOG FOR MEDIA FILE
506
TRANSMIT SUBSEQUENT MESSAGE TO LOG SERVER WHILE EMBEDDED MEDIA PLAYER PAGE OPEN
508
RECORD TIME STAMP FOR SUBSEQUENT MESSAGE IN LOG FOR MEDIA FILE

FIG. 5

the media file (such as a unique identifier of the media file used in the media fil[e] index and log system 122 and received along with a search results page) . . . and the ranking (if any) of the media file within the search results page, and a session identifier indicating the communications thread between client 110 and customer system 118." *Id.* ¶0058.

59. The media file index and log system 122 in *Hayward* "maintains a

respective log for each indexed media file." *Id.* The media file index and log system 122 "records that the media file has been selected for playing by a user." *Id.* In particular, at step 504, the media file index and log system 122 stores "the information contained within the media file identification message in the media file's respective log," including a "time stamp" that identifies "the time at which the media file identification message was transmitted." *Id.*

60. *Hayward* teaches that "[t]he embedded media player page also preferably includes scripting to instruct the client 110 to transmit at least one subsequent message at step 506 while the embedded media player page remains open." *Id.* ¶0060. As noted above, *Hayward* defines "scripting" as "server or client-side programming which supplements a static HTML page." *Id.* ¶0017. *Hayward* contemplates that the "scripting" could be, for example, "Java" or "JavaScript." *Id.* As shown at step 506, the subsequent messages are transmitted to "the log server of the media file index and log system 122." *Id.* ¶0061. The subsequent messages are transmitted "at predetermined time intervals while the embedded media play page remains open," such as "every thirty seconds." *Id.* ¶¶0060–0061. At step 508, a time stamp for the at least one subsequent message is stored in the log associated with the media file." *Id.* ¶0060.

61. As noted above, "[t]he media file index and log system 122 preferably indexes and maintains logs for a plurality of media files." *Id.* ¶0062. "When the

logs maintained by the media file index and log system 122 are populated with sufficient data," *Hayward* teaches, "this data can be processed . . . to provide valuable information," including "raw popularity data" that "allows for the ranking of the popularity of media files that are indexed in the media file index and log system 122, based at least in part on the ranking results." *Id.*

62. *Hayward* recognizes that "much information can be gleaned about the user and the playing event" from the time stamps stored in a media file's log at the media file index and log system 122. *Id.* ¶0063. In particular, *Hayward* teaches, "by calculating the difference in time between the first and last time stamps for a media file during a selected playing session recorded in the log, the approximate length of time that the embedded media player page was left open by the user can be calculated." *Id.* *Hayward* recognizes that this information may be valuable to, for example, an advertiser whose advertisement was displayed while the embedded media player page was open. *Id.* ¶0064. This information may also determine "how pertinent or relevant a played media file was to a user's initial search request." *Id.*

B. *Middleton*

63. I note that *Middleton* was published on August 15, 2002, which was more than one year before August 21, 2008. I am told that makes *Middleton* prior art to the '609 patent under § 102(b).

64. *Middleton* describes an applet, downloaded to a user's web browser, that tracks the user's interactions with an object on a web page. *Middleton*, Abstract. The applet tracks, for example, a "time [the object is] displayed on [the] page." *Middleton*, Abstract; *see also id.* ¶0037.

65. The applet in *Middleton* takes the form of "Java^{TM} code 44 that includes instructions to be run while [a] user computer 20a is displaying the web page." *Id.* ¶0026. In particular, the Java^{TM} code 44 "includes an applet program and data for tracking and logging the activities of the user in memory 24 while the user is viewing the Web page." *Id.* ¶0029. *Middleton*'s applet "permits the authors of the advertisement 39 to better understand how the users interact with the Web page advertisement." *Id.* ¶0029.

66. Figure 2, below right, shows how the applet allows an advertiser to track "the elapsed time that [an] element 48 [of the advertisement 39] has been displayed on the page." *Id.* ¶0037. In state 104, *Middleton* teaches, "user activities

with respect to objects within the advertisement 39 may begin to be tracked by logging information in local memory locations 24 at the client 20." *Id.* ¶0036. The "elapsed time" is tracked in state 106. When "state 120 is entered . . . the activity log 60 is sent from the local memory 24 by the applet 44 back to a server," which "may or may not be the same server [] from which the Web page 46 was originally downloaded." *Id.* ¶0045.

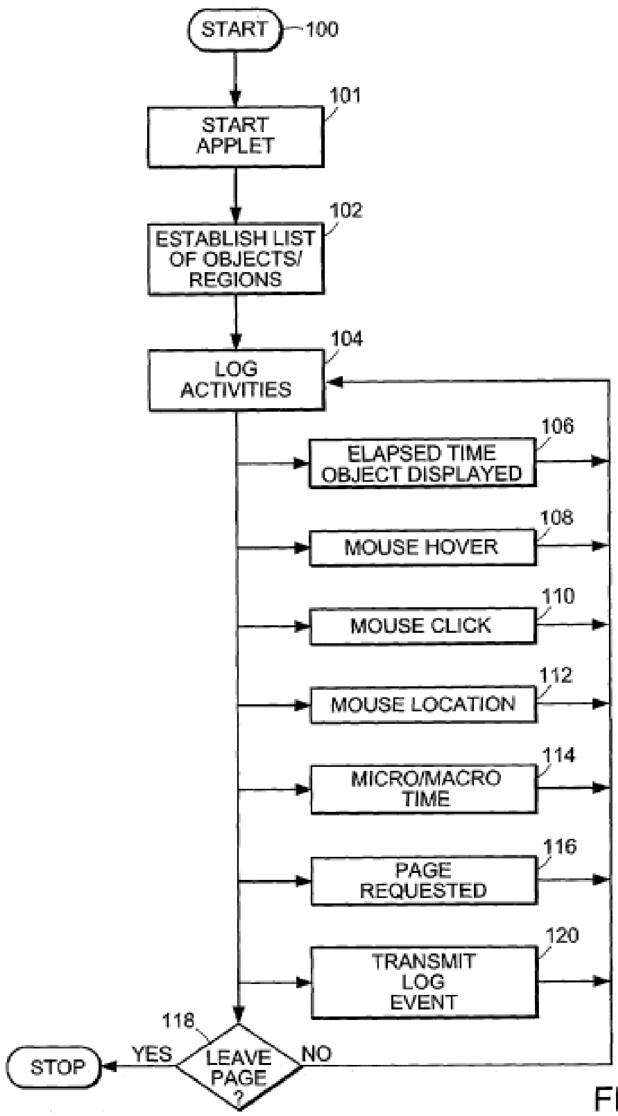


FIG. 2

67. According to *Middleton*, this information may be valuable to an advertiser, who may wish to understand "what motivates users to pay initial attention to and/or otherwise interact with Web page advertising." *Id.* ¶0010.

C. *Ryan*

68. I note that *Ryan* issued on July 16, 2002, which was more than one year before August 21, 2008. I am told that makes *Ryan* prior art to the '609 patent

under § 102(b).

69. *Ryan* describes a search engine that includes a server and a database. *Ryan*, 1:23–29. In *Ryan*, when a user enters a search command at a personal computer, the server receives the search command, uses it to search the database, and provides search results, such as a list of web pages, to the user's personal computer for display to the user. *Id.* at1:23–30. The search engine could be used to provide, for example, videos. *Id.* at 36:64–67.

70. A problem with search engines, *Ryan* notes, is that they fail to take into account "any measure of the actual users' opinions" regarding the search results, even though it would "directly benefits the advertiser, because it allows for content to be targeted in real time based upon various criteria." *Id.* at 1:66–2:4, 4:57–60. Accordingly, *Ryan* proposes determining a "relevance" to the user of a web page selected from the search results. *Id.* at 9:17–18.

71. As *Ryan* explains, "[d]epending on the relevance of the site, the user may spend time reading, downloading exploring further pages, embedded links and so forth, or if the site appears irrelevant/uninteresting, the user may return directly back to the search results after a short period." *Id.* at 9:17–22. So *Ryan* uses a "Java applet" to record a "date-time" when the user selects a site. *Id.* at 8:63–67, 9:41–56. "The time difference between the two selections," *Ryan* teaches, "is recorded as the difference between two . . . time data 132 from subsequent

selections from the list of web page searches." *Id.* at 9:22–25. This time difference is recorded as "surfer trace data on the popularity of web pages." *Id.* at 9:29–30.

72. From the surfer trace data, the server generates a "cumulative surfer trace table," shown below. *Id.* at 13:62–14:3. The cumulative surfer trace table is updated each time a user selects a web page from the search results. *Id.* at 16:10–16.

TABLE 4

Each row is one surfer trace and the combined rows are the cumulative surfer trace

IP Number	User ID	Keyword	URL (webpage)	Date-time

73. *Ryan*'s server also maintains a table, shown below, linking web pages with keywords entered in search commands. *Id.* at 12:16–41. This table includes "the cumulative number of significant visits (hits) to each URL addresses corresponding to each key-word," which *Ryan* calls "weighting factor X." *Id.* at 12:27–29. The weighting factor X "is a measure of the popularity of the URL for each keyword and is determine [sic] from the surfer traces." *Id.* at 12:29–31. The weighting factor X may be "increment[ed] . . . based on the time spent at the web page," *Ryan* teaches. *Id.* at 16:40–41. "The longer the time spent the more this increments the value of X." *Id.* at 16:41–42.

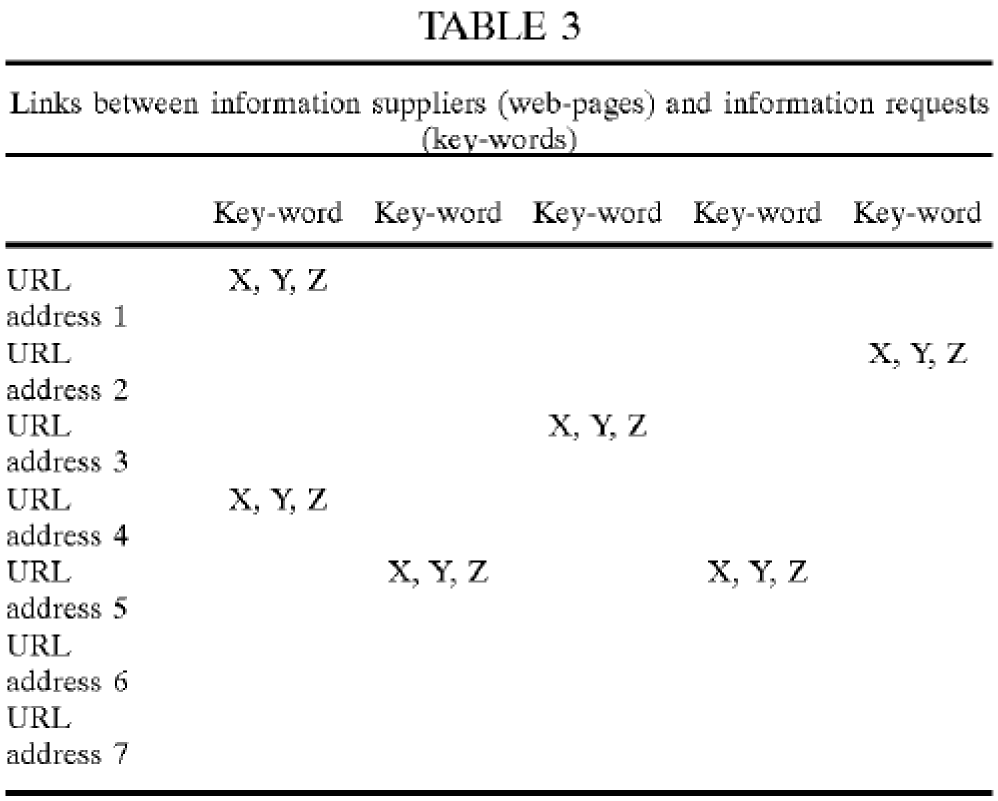


TABLE 3

Links between information suppliers (web-pages) and information requests (key-words)

	Key-word	Key-word	Key-word	Key-word	Key-word
URL address 1	X, Y, Z				
URL address 2					X, Y, Z
URL address 3			X, Y, Z		
URL address 4	X, Y, Z				
URL address 5		X, Y, Z		X, Y, Z	
URL address 6					
URL address 7					

VIII. Claims 1–3 of the '609 Patent are Unpatentable

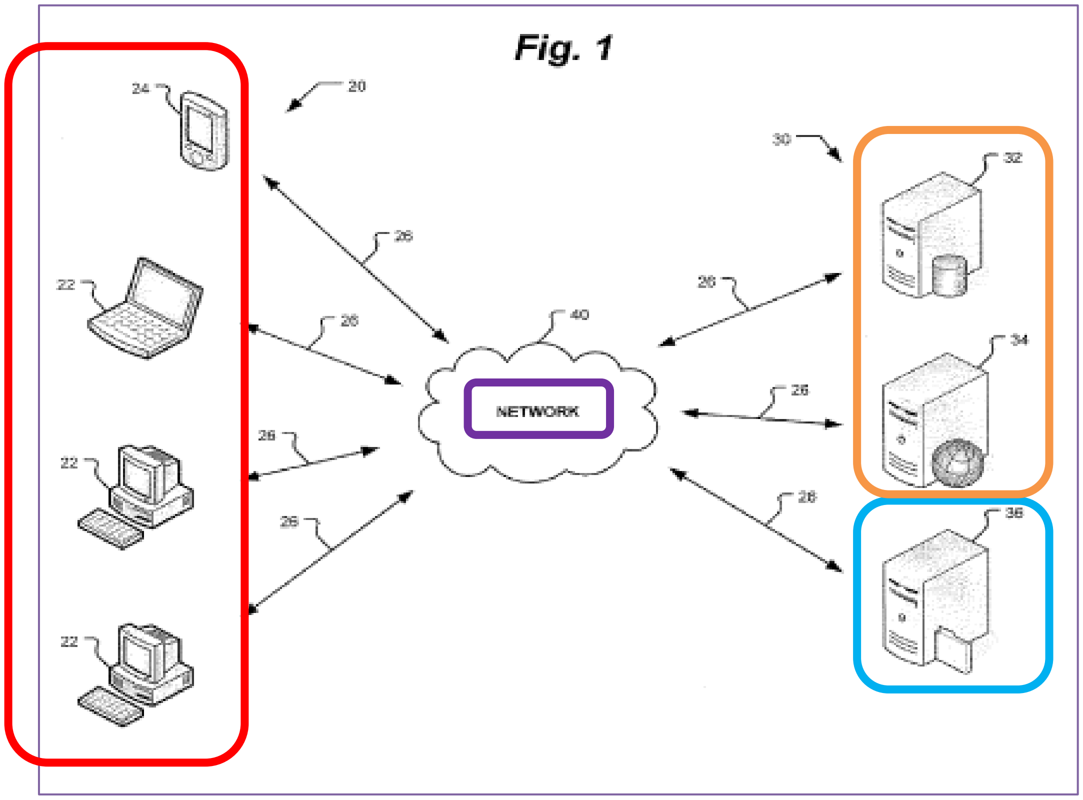
A. Ground 1: *Hayward* anticipates claim 1

1. Claim 1

a. [1a] "A method for tracking digital media presentations delivered from a first computer system to a user's computer via a network comprising:"

74. In my opinion, a POSA would have understood *Hayward* to disclose this element.

75. I understand the "user's computer" to refer to the user computers 20 shown in Figure 1, annotated below. I understand the "first computer system" to be the content or web server 34 and database server 32, which are connected to the user computers 20 via the network 40.



76. *Hayward* discloses “a method of displaying video data using an embedded media player page.” *Hayward*, Abstract. The embedded media player page is provided by a “customer system” to a “client” where the video data is displayed. *Id.* ¶0031. *Hayward*’s “embedded media player page also facilitates the collection of data in connection with the playing of a media file.” *Id.* ¶0057. This data is collected from the client by a “media file index and log system.” *Id.* ¶0058.

77. In *Hayward*, the method is implemented in a “system of interconnected computer system networks,” as shown in Figure 1, annotated below. *Id.* ¶0019. Each of the computer system networks 102, such as that labeled in orange, is connected to the client 110 via the Internet 114. *Id.*

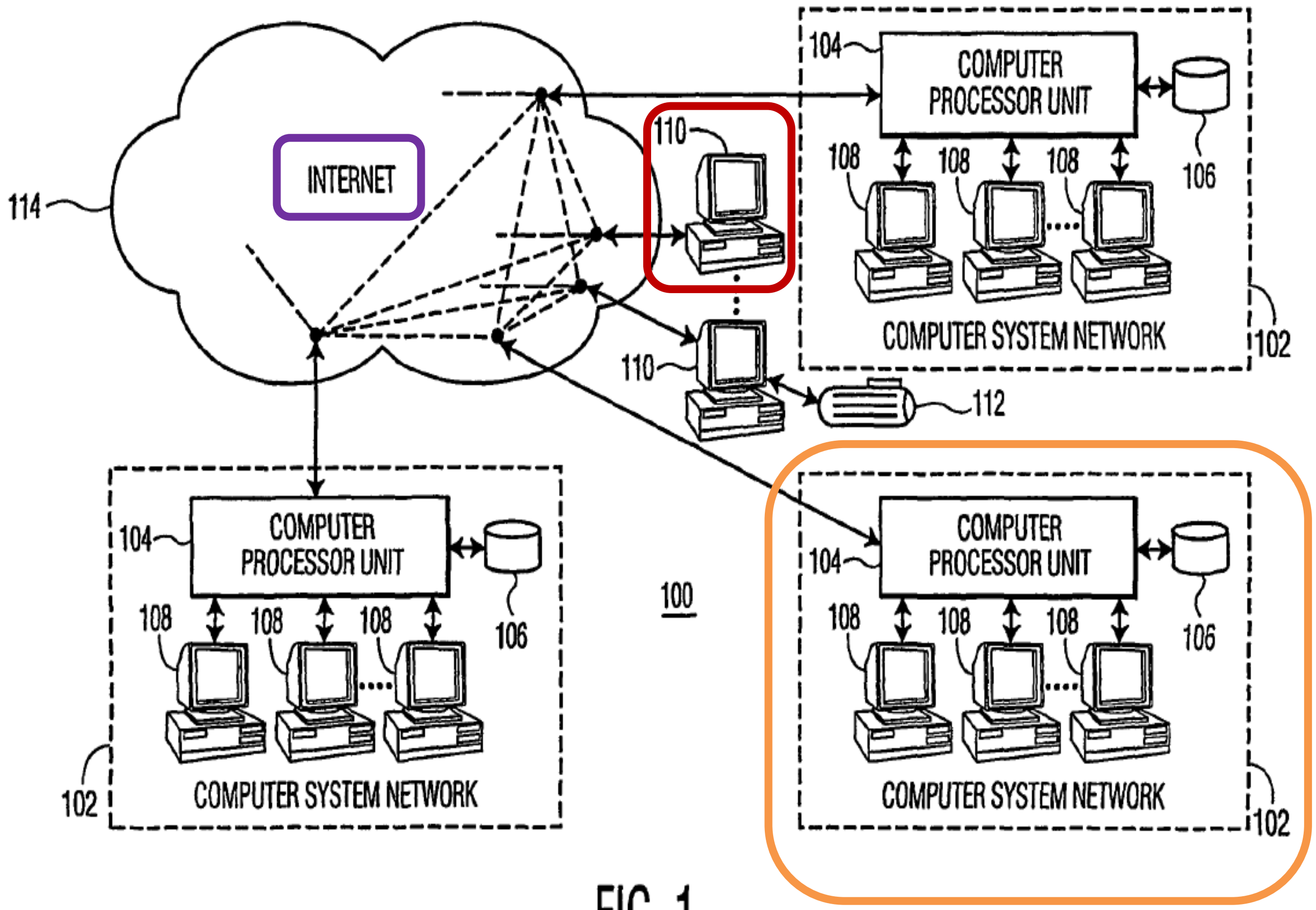


FIG. 1

78. In my opinion, *Hayward* teaches that one of the computer systems 102 in Figure 1, such as that labeled in orange above, may include both a customer system 118 and media file index and log system 122, as shown in Figure 1A, annotated below. *Id.* ¶0024. In particular, *Hayward* teaches that each of customer system 118 and media file index and log system 122 may take the form of a computer system 102, and *Hayward* teaches that these systems may be "combined physically within one . . system[]" in some embodiments. *Id.* ¶0030.

79. *Hayward* also teaches that the first computer system (customer

system 118 and media file index and log system 122) and the user's computer (client 110) are connected via a network (internet 114).

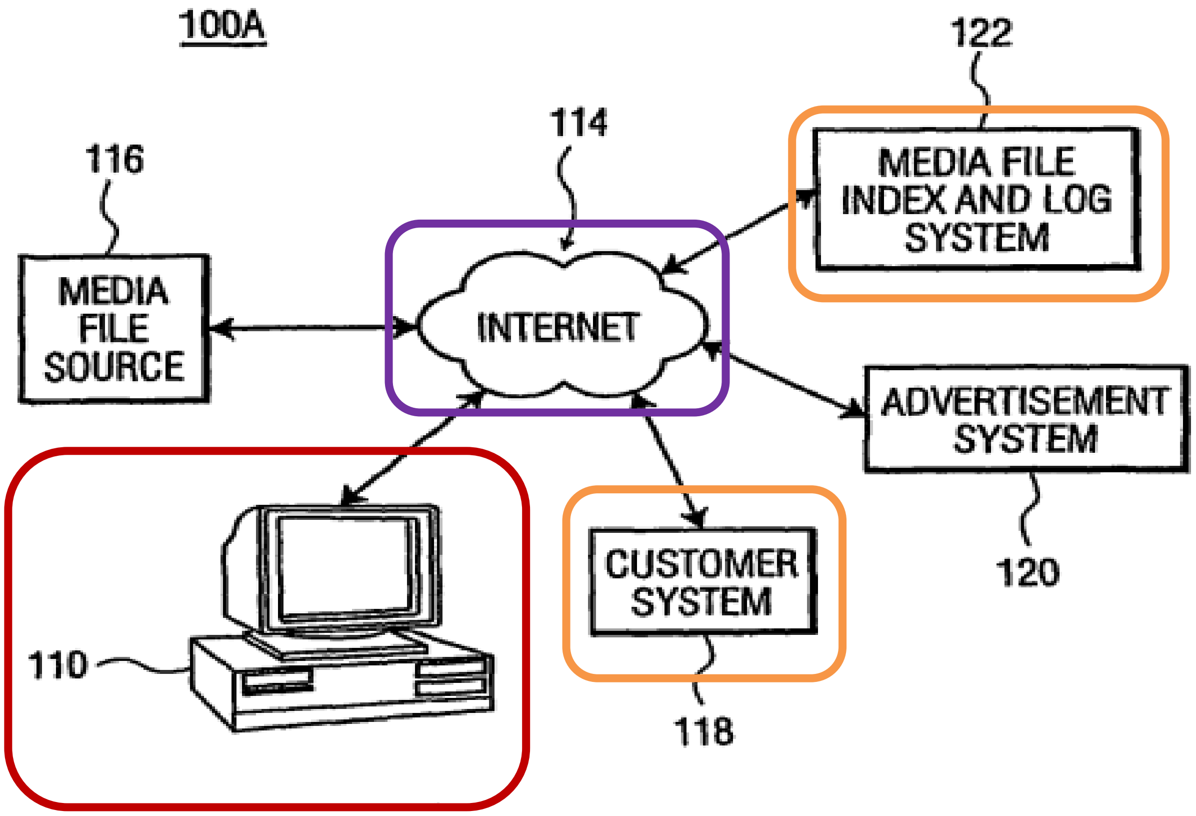


FIG. 1A

80. In my opinion, a POSA would have understood *Hayward*'s customer system 118 and media file index and log system 122 to disclose the claimed "first computer system"; would have understood *Hayward*'s client 110 to disclose the claimed "user's computer"; and would have understood *Hayward*'s Internet 114 to disclose the claimed "network." Further, because *Hayward* teaches a method for tracking a media file displayed using an embedded media player page provided by

the customer system 118 to the client 110 via the Internet 114, in my opinion *Hayward* discloses "[a] method for tracking digital media presentations delivered from a first computer system to a user's computer via a network."

b. [1b] "providing a corresponding web page to the user's computer for each digital media presentation to be delivered using the first computer system;"

81. In my opinion, *Hayward* discloses providing a corresponding web page (e.g., the embedded media player page) to the user's computer (e.g., the client 110) for each digital media presentation to be delivered using the first computer system (e.g., the customer system 118).

82. *Hayward* teaches that a user can search for media files by providing a search request to the customer system 118 through the client 110. *Hayward*, ¶0028. The customer system 118 receives the search request and transmits it to the media file index and log system 122, which "searches local and remote databases for indexed media files that satisfy the search request" to generate "search results." *Id.* The "search results" include "a database identifier unique to each video file." *Id.* The media file index and log system 122 provides the search results to the customer system 118, and the customer system 118 provides the search results to the client 110 for display to the user. *Id.*

83. When the user selects one of the search results, the client 110 requests the embedded media player page from the customer system 118. *Id.* ¶0029. The

embedded media player page is then “transmitted from customer system 118 through Internet 114 and displayed to the user by a client 110.” *Id.* ¶0031.

84. The embedded media player page is “displayed as a Web page in a browser window” at the client 110 and “includes a reference to a functional media player object” to play the selected media file. *Id.* ¶¶0024, 0031. An example embedded media player page is shown in Figure 2 of *Hayward*, below. As shown, the embedded media player page “includes video display area 202 (when the embedded player plays video files).” *Id.* ¶0032.

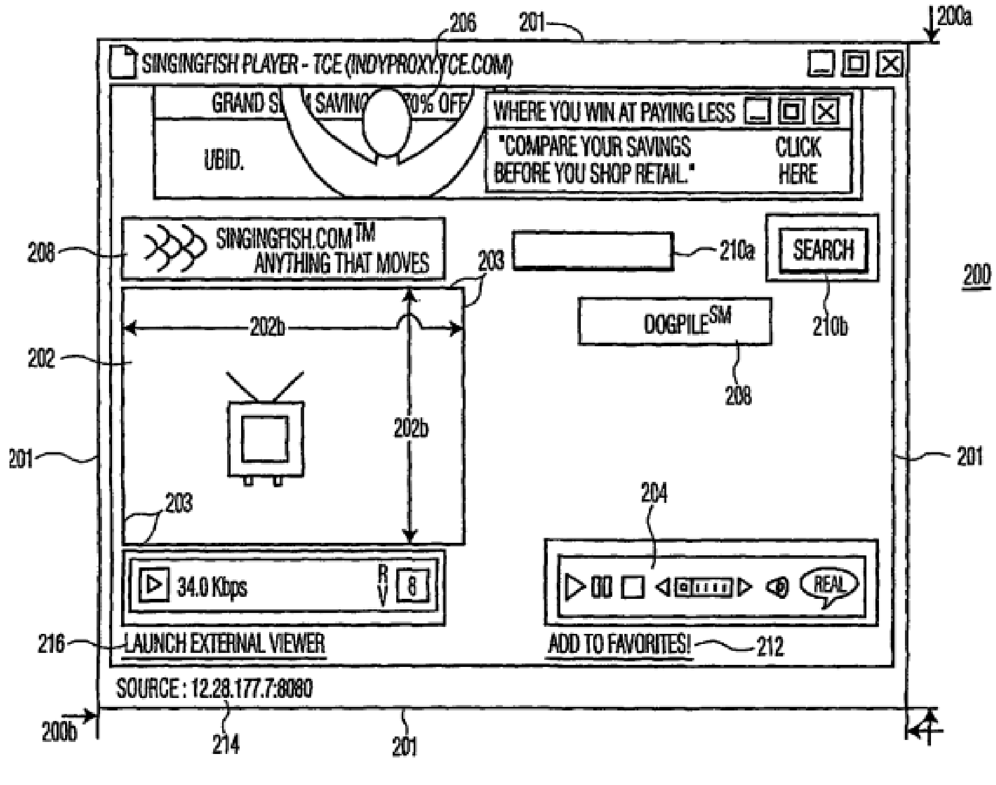


FIG. 2

85. In my opinion, *Hayward*’s embedded media player page discloses the claimed “corresponding web page.” Because *Hayward* discloses that customer

system 118 provides the embedded media player page to client 110, and that the embedded media player page facilitates display of the media file selected by the user, in my opinion *Hayward* discloses "providing a corresponding web page to the user's computer for each digital media presentation to be delivered using the first computer system."

c. [1c] "providing identifier data to the user's computer using the first computer system"

86. In my opinion, *Hayward* discloses providing identifier data (e.g., a unique identifier for the media file and a session identifier) to the user's computer (client 110) using the first computer system (customer system 118 and media file index and log system 122).

87. *Hayward* teaches that "[a] user of client 110 accesses customer system 118 through Internet 114," and "customer system 118 transmits a web page to client 110 through Internet 114." *Hayward*, ¶0025. The transmitted web page includes "a media file search prompt" where the user may enter a search request, such as "Pearl Harbor" and "movie trailer." *Id.* ¶¶0026, 0028. "The search request is received by customer system 118 and is transmitted to media file index and log system 122." *Id.* ¶0028. When the search request is transmitted by the customer system 118, the media file index and log system 122 searches "for indexed media files that satisfy the search request" and "transmits the results to customer system 118." *Id.* ¶0028. The search results include "the playing length of each

video file, the URI address of each video file, encoding bit rate of the video file, file format, a database identifier unique to each video file, frame dimensional data for each video file, or any other information contained within the database." *Id.* ¶0028. This "unique identifier for the media file" is "associated with the media file" and used to identify the media file in the media file index and log system 122. *Id.* ¶¶0027, 0054, 0058, Fig. 5. In my opinion, the claimed "identifier data" could include *Hayward*'s unique identifier for the media file.

88. *Hayward* further teaches that the user may "view the video data contained within a video file listed in the search results displayed to the user by clicking a link to one of the video files." *Id.* ¶0029. When the user clicks a link for a selected media file, "the customer system 122 instructs the client to request [an] embedded media player page from the customer system 122." *Id.* A POSA would have understood from *Hayward* that, in connection with the providing the embedded media player page to the client 110, the customer system 122 would provide the client 110 with a session identifier. *Hayward* references a "session identifier indicating the communication thread between client 110 and customer system 118" that the client 110 sends in the media file identification message. *Id.* ¶0058. A POSA would have understood that, in order for the client 110 to send this session identifier in the media file identification message, the client 110 would have had to receive it from the customer system 118 in connection with the

embedded media player page. In my opinion, the claimed "identifier data" could include *Hayward*'s session identifier.

89. Because *Hayward* thus discloses providing, for example, a unique identifier for the media file and a session identifier to the client 110 using the customer system 118, in my opinion *Hayward* discloses "providing identifier data to the user's computer using the first computer system."

d. [1d] "providing an applet to the user's computer for each digital media presentation to be delivered using the first computer system, wherein the applet is operative by the user's computer as a timer"

90. In my opinion, *Hayward* discloses providing an applet ("scripting," as defined in *Hayward*) to the user's computer (client 110) for each digital media presentation (video file) to be delivered using the first computer system (customer system 118 and media file index and log system 122), wherein the applet is operative by the user's computer (client 110) as a timer.

91. In *Hayward*, when the user clicks a link for a selected media file, "the customer system 122 instructs the client to request [an] embedded media player page from the customer system 122," and the media player displays the selected media file. *Hayward*, ¶¶0029, 0046.

92. According to *Hayward*, the embedded media player page also "facilitates the collection of data in connection with the playing of the video file." *Id.* ¶0057. In particular, "the embedded media player page instructs the client 110

to transmit a media file identification message to [the] media file index and log server 122." *Id.* ¶0058. Additionally, the embedded media player page "instruct[s] the client 110 to transmit at least one subsequent message . . . while the embedded media player page remains open." *Id.* ¶0060.

93. *Hayward* states that the embedded media player page instructs the client 110 to send the subsequent messages using "scripting." *Id.* *Hayward* defines "scripting," as used in *Hayward*, as "server or client-side programming which supplements a static HTML page." *Id.* ¶0017. *Hayward* contemplates that its "scripting" could be, for example, "Java" or "JavaScript." *Id.* In my opinion, a POSA would have understood that "client-side programming" in "Java" that "supplements a static HTML page" includes applets. Accordingly, in my opinion, a POSA would have understood *Hayward* to disclose that the embedded media player page uses an applet to instruct the client 110 to send the subsequent messages.

94. Just as *Hayward* states that "[t]he embedded media player page . . . instruct[s] the client 110 to transmit at least one subsequent message," *Hayward* states that "the embedded media player page instructs the client 110 to transmit [the] media file identification message." *Id.* ¶¶0058, 0060. But while *Hayward* states that the embedded media player page instructs the client 110 to send the subsequent messages using "scripting," *Hayward* does not state how the embedded

media player page instructs the client 110 to send the media file identification message. *Id.* In my opinion, a POSA would have understood from *Hayward* that, as with the subsequent messages, the embedded media player page uses "scripting" to instruct the client 110 to send the media file identification message. As noted above, *Hayward* contemplates that its "scripting" could be, for example, "Java" which a POSA would have understood to include applets. *Id.* ¶0017. Accordingly, in my opinion, a POSA would have understood *Hayward* to disclose that the embedded media player page uses an applet to instruct the client 110 to send the media file identification message.

95. In my opinion, the scripting in *Hayward* that instructs the client 110 to send the media file identification message and the subsequent messages discloses the claimed "applet." Like the claimed applet, which is "provid[ed] . . . to the user's computer for each digital media presentation to be delivered using the first computer system," *Hayward*'s scripting is provided to the client 110 in each embedded media player page through which a selected media file is displayed. Further, like the claimed applet, which is "operative by the user's computer as a timer," *Hayward*'s scripting is operative by the client to instruct the client to send the subsequent messages "at predetermined time intervals." *Id.* ¶0060; *see also id.* ¶0061 ("periodic intervals," "every thirty seconds").

96. Because a POSA would have understood *Hayward* to disclose

providing an applet to client 110 in each embedded media player page and using the applet to instruct the client 110 to send the media file identification message and the subsequent messages to the media file index and log system 122 at predetermined time intervals, in my opinion a POSA would have understood *Hayward* to disclose "providing an applet to the user's computer for each digital media presentation to be delivered using the first computer system, wherein the applet is operative by the user's computer as a timer."

e. [1e] "receiving at least a portion of the identifier data from the user's computer responsively to the timer applet each time a predetermined temporal period elapses using the first computer system"

97. In my opinion, *Hayward* discloses receiving at least a portion of the identifier data (the unique identifier for the media file) from the user's computer (client 110 in *Hayward*) responsively to the timer applet ("scripting") each time a predetermined temporal time period elapses using the first computer system (customer system 118 and media file index and log server 122).

98. As noted above in connection with [1c], in my opinion the claimed "identifier data" could include *Hayward*'s unique identifier for the media file. Accordingly, in my opinion, the unique identifier for the media file also discloses the claimed "at least a portion of the identifier data."

99. In *Hayward*, "the embedded media player page instructs the client 110 to transmit a media file identification message to [the] media file index and log

server 122." *Hayward*, ¶0058. *Hayward* states that the media file identification message "should at least identify to the log system 122 the media file that is to be played by the embedded media player page." *Id.* For example, *Hayward*'s media file identification message may include the "unique identifier for the media file (such as a unique identifier of the media file used in the media fil[e] index and log system 122 and received along with the search results page." *Id.*

100. *Hayward*'s client 110 further sends the "subsequent messages" to the media file index and log system 122. *Id.* ¶0061. In my opinion, a POSA would have understood that the subsequent messages, like the media file identification message, would have uniquely identified the media file. This is because *Hayward*'s media file index and log system 122 indexes "a plurality of media files," each of which is identified by "a unique identifier for the media file," and "maintains a respective log for each indexed media file." *Id.* ¶¶0027, 0058. When a subsequent message is received in *Hayward*, "a time stamp for the . . . subsequent message is stored in the log associated with the media file." *Id.* ¶0060. In my opinion, a POSA would have understood that, to store the time stamp in the "log associated with the media file," the subsequent message would need to uniquely identify the media file, like the media file identification message does. This similarity between the subsequent messages and the media file identification message is consistent with *Hayward*'s description of each as "an HTTP request to

the media file index and log system 122 for a one-pixel GIF file." *Id.* ¶¶0059, 0061. Just as the unique identifier for the media file is "appended to the HTTP request" that is the media file identification message, a POSA would have understood *Hayward* to teach that the unique identifier for the media file is appended to the HTTP requests that are the subsequent messages. *Id.* This would have allowed the time stamps for the subsequent messages to be "stored in the log associated with the media file" by the media file index and log system 122, as *Hayward* describes. *Id.* ¶0060.

101. As discussed above, in my opinion *Hayward* discloses using an applet to instruct the embedded media player page to send the subsequent messages to the media file index and log system 122 at predetermined time intervals. Accordingly, in my opinion, *Hayward* discloses that the media file index and log system 122 receives the unique identifier for the media file from the client 110 responsively to the applet each time a predetermined temporal period elapses.

102. Because a POSA would have understood *Hayward* to disclose the media file index and log system 122 receiving the subsequent messages including the unique identifier for the media file sent using the applet at the client 110 each time a predetermined temporal time period elapses, in my opinion a POSA would have understood *Hayward* to disclose "receiving at least a portion of the identifier data from the user's computer responsively to the timer applet each time a

predetermined temporal period elapses using the first computer system."

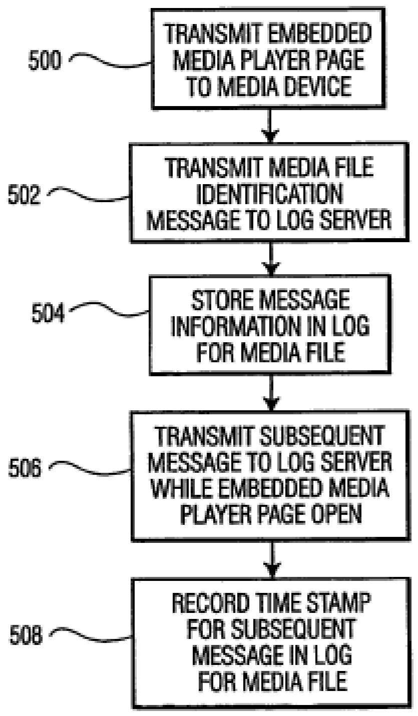
f. [1f] "storing data indicative of the received at least portion of the identifier data using the first computer system"

103. In my opinion, *Hayward* discloses storing data (entries at the media file index and log system 122 relating to the media file information message and subsequent messages) indicative of the received at least portion of the identifier data (unique identifier for the media file) using the first computer system (customer system 118 and media file index and log system 122).

104. In *Hayward*, the "media file index and log system 122 preferably maintains a respective log for each media file." *Hayward*, ¶0058. When the media file identification message is received, the media file index and log system 122 "records that the media file has been selected for playing by a user, preferably by storing, at step 504 [in Figure 5, below right] the information contained within the media file identification message in the media file's respective log." *Id.* This information includes the unique identifier for the media file, a "time stamp . . identifying the time at which the media file identification message was

transmitted," and "the time that the media file identification message was received." *Id.*

105. Similarly, *Hayward* discloses that a time stamp for each subsequent message is "stored in the log associated with the media file," as shown in Figure 5, right. *Id.* ¶0060. As noted above in connection with [1e], in my opinion a POSA would have understood that, to store the time stamp in the "log associated with the media file," as *Hayward* contemplates, the subsequent message would need to uniquely identify the media file. In my opinion, a POSA would have understood that the stored time stamps for the subsequent messages in *Hayward* are indicative of the unique identifier for the media file because their storage in the "log associated with the media file" indicates the media file.


500
TRANSMIT EMBEDDED MEDIA PLAYER PAGE TO MEDIA DEVICE
502
TRANSMIT MEDIA FILE IDENTIFICATION MESSAGE TO LOG SERVER
504
STORE MESSAGE INFORMATION IN LOG FOR MEDIA FILE
506
TRANSMIT SUBSEQUENT MESSAGE TO LOG SERVER WHILE EMBEDDED MEDIA PLAYER PAGE OPEN
508
RECORD TIME STAMP FOR SUBSEQUENT MESSAGE IN LOG FOR MEDIA FILE
FIG. 5

106. Because *Hayward* discloses storing the unique identifier for the media file and the time stamp for the media file identification message, as well as the time stamps for the subsequent messages, in the log associated with the media file at the media file index and log system 122, in my opinion *Hayward* discloses

"storing data indicative of the received at least portion of the identifier data using the first computer system."

g. [1g] "wherein each provided webpage causes corresponding digital media presentation data to be streamed from a second computer system distinct from the first computer system directly to the user's computer independent of the first computer system"

107. In my opinion, *Hayward* discloses wherein each provided webpage (embedded media player page) causes corresponding digital media presentation data (media file) to be streamed from a second computer system (media file source 116) distinct from the first computer system (customer system 118 and media file index and log system 122) directly to the user's computer (client 110) independent of the first computer system.

108. In *Hayward*, the media file is streamed directly to the client 110 by a media file source 116, as shown in annotated Figure 1A below. Media file source 116 "is accessible through Internet 114 and provides at least one media file through Internet 114 for playing on client 110." *Hayward*, ¶0021.

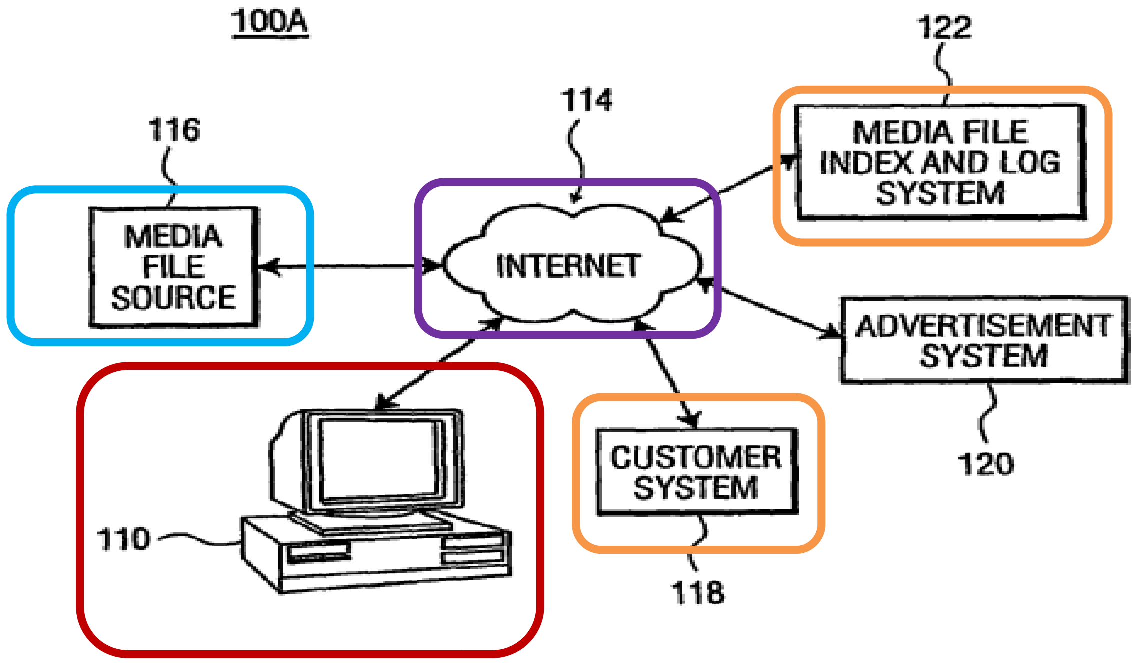


FIG. 1A

109. *Hayward* describes an example in which a media file, "thestream.asx," is streamed to the client 110 from a media file source 116, "thestreamhost.com." *Id.* ¶¶0042–0046. When a user selects a search result in the embedded media player page, "[a] file entitled 'thestream.asx' is found and streamed by the media player from [the] media file source 116 located at 'thestreamhost.com' through Internet 114 via a SRC command." *Id.* ¶0046. "The file 'thestream.asx'," *Hayward* discloses, "is outputted by the embedded medial

player.” *Id.*

110. In my opinion, a POSA would have understood from *Hayward*’s example that the media file, “thestream.asx,” is streamed from the media file source 116 directly to the client 110 independent of the customer system 118 and media file index and log system 122. “[T]he embedded media player reference[s]” the media file, “thestream.asx,” using, for example, a “SRC command”:

SRC=“http://thestreamhost.com/thestream.asx”

Id. ¶¶0044, 0046.

111. In my opinion, a POSA would have understood that a SRC command causes a web browser at a client (here, the client 110) to fetch a media file (here, “thestream.asx”) from a content source (here, media file source 116) located at the Internet domain or address given in the universal resource locator (“URL”) specified by the command (here, http://thestreamhost.com/thestream.asx). A POSA would have understood that the URL in the SRC command may specify any domain or address on the Internet and has no relation to a content server providing the web page in which the media file is ultimately embedded and displayed. Thus, a POSA would have understood that *Hayward* encompasses an embodiment in which the media file is streamed from the media file source 116 directly to the client 110 independent of the customer system 118 and media file index and log system 122.

112. In my opinion, a POSA would also have understood *Hayward* to encompass an embodiment in which the media file source 116 is distinct from the customer system 118 and media file index and log system 122, as shown in Figure 2. *Hayward* discloses that the media file source 116, like the customer system 118 and media file index and log system 122, may be "connected to Internet 114 and may be configured as [a] computer system network 102," shown in Figure 1 to be distinct. *Id.* ¶0024. In my opinion, a POSA would have understood from *Hayward*'s SRC command that the media file source 116, to which the URL points, may be a second computer system distinct from the first computer system (customer system 118 and media file index and log system 122). In particular, when embedding content in a Web page with an HTML SRC directive, there is no requirement that the source URL shares a domain with the source of the containing page, or that the two content sources are controlled or operated by the same party: they may be unrelated.

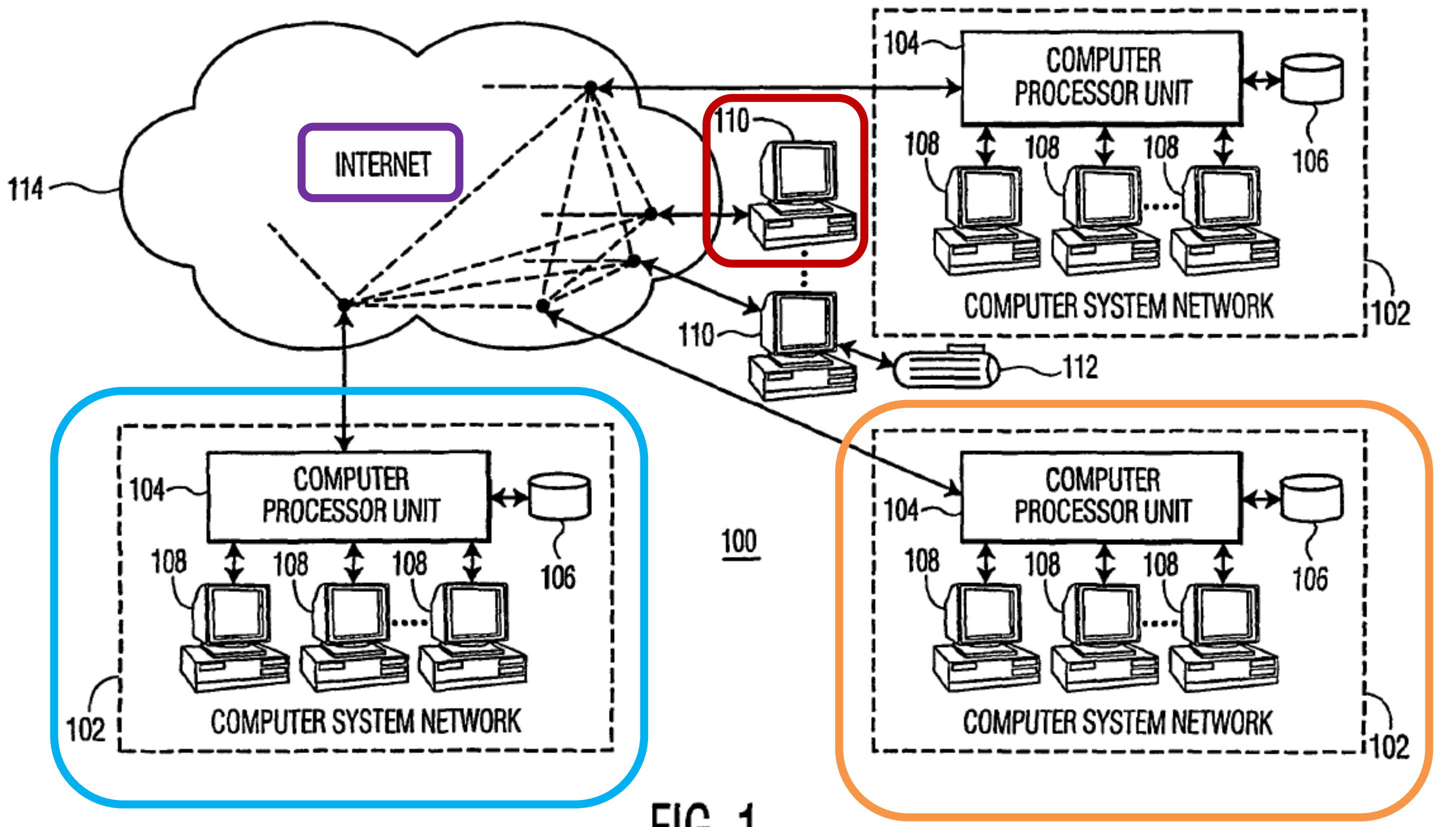


FIG. 1

113. Because *Hayward* encompasses an embodiment in which the embedded media player page causes the media file to be streamed from the media file source 116 directly to the client 110 independent of the customer system 118 and media file index and log system 122, and in which *Hayward*'s media file source 116 is distinct from the customer system 118 and media file index and log system 122, in my opinion *Hayward* discloses "wherein each provided webpage causes corresponding digital media presentation data to be streamed from a second computer system distinct from the first computer system directly to the user's

computer independent of the first computer system."[1]

h. [1h] "wherein the stored data is indicative of an amount of time the digital media presentation data is streamed from the second computer system to the user's computer"

114. In my opinion, *Hayward* discloses that the stored data (entries at the media file index and log system 122 relating to the media file identification message and the subsequent messages) is indicative of an amount of time the digital media presentation data (media file) is streamed from the second computer system (media file source 116) to the user's computer (client 110).

[1] I have been told that Google has proposed construing "a second computer system distinct from the first computer system" as a second computer system unrelated to the first computer system and not commonly controlled or operated by the same party," while Uniloc has proposed a plain and ordinary meaning. Ex. 1008 at 10; Ex. 1009 at 2. In my opinion, *Hayward* discloses [1h] under either of these constructions. As noted above, *Hayward* encompasses an embodiment in which the media file source 116 in *Hayward* is unrelated to the customer system 118 and media file index and log system 122. *Hayward*, ¶0024, Fig. 1. *Hayward* also discloses an embodiment in which the media file source 116, on the one hand, and the customer system 118 and media file index and log system 122, on the other, are not commonly controlled or operated by the same party. *Id.* ¶0024.

115. As discussed above in [1f], *Hayward*'s media file index and log system 122 stores a "time stamp . . . identifying the time at which the media file identification message was transmitted," and "the time that the media file identification message was received" in a log associated with the media file. *Hayward*, ¶0058. Similarly, *Hayward* discloses that a time stamp for each subsequent message is "stored in the log associated with the media file." *Id.* ¶0060.

116. In my opinion, a POSA would have understood from *Hayward* that the "time stamp . . identifying the time at which the media file identification message was transmitted" in *Hayward* is indicative of a time a streaming file is buffered and begins to play. This is because in *Hayward* "[t]he media file identification message is preferably transmitted to the media file index and log system 122 substantially proximate in time to when the media file begins to play in the embedded media player of the embedded media player page, particularly in the case of streaming media files which are typically buffered for a period of time before play begins." *Id.* ¶0059.

117. Further, in my opinion, a POSA would have understood from *Hayward* that the time stamps for the subsequent messages in *Hayward* are indicative of a time the embedded media player page is open once the media file begins to play. The subsequent messages are sent "at predetermined time intervals

while the embedded media player page remains open." *Id.* ¶0060. As *Hayward* explains, "by calculating the time difference between the first and last time stamps for a media file during a selected playing session recorded in the log, the approximate length of time that the embedded media player page was left open by the user can be calculated." *Id.* ¶0063.

118. Still further, in my opinion, a POSA would have understood from *Hayward* that the time stamps for the subsequent messages in *Hayward* are indicative of a time a streaming media file is played. This is because *Hayward* encompasses embodiments in which the embedded media player page is open at the client 110 for an amount of time that is less than or equal to the playing time of the media file. *Id.* A POSA would have understood that, in these embodiments, each time stamp of a subsequent message would indicate not only that the embedded media player page was open, but also that the media file was being played.

119. Finally, in my opinion, a POSA would have understood from *Hayward* that the time stamps for the media file identification message and the subsequent messages are indicative of an amount of time the media file data is streamed from the media file source 116 to the client 110. I have been told that Google has proposed construing "is indicative of an amount of time the digital media presentation data is streamed from the second computer system to the user's

computer" as "equates to the amount of time that a digital media presentation data is transferred as a substantially steady and continuous stream from the second computer," while Uniloc has proposed giving this term its plain and ordinary meaning. Ex. 1008 at 7; Ex. 1009 at 2. In my opinion, *Hayward* discloses [1h] under either of these constructions. Regarding Google's construction, in my opinion a POSA would have understood that *Hayward* encompasses embodiments in which the entirety of a media file is streamed to and displayed by the user's computer. A POSA would further have understood that the time stamps in these embodiments would equate to the amount of time that the media file was transferred as a substantially steady and continuous stream from the media file source 116. This is because, in such embodiments, the time that the media file was transferred would equate to the time that the media file was displayed. *Hayward*, ¶0059. A POSA would have understood that, in *Hayward*'s system, the transferring of the media file as a substantially steady and continuous stream will precede the display of the media file by a buffering window. As *Hayward* explains, "streaming media files . . . are typically buffered for a period of time before play begins." *Id.* A POSA would further have understood that, in embodiments where the entirety of the media file is streamed to and displayed by the user's computer, the display of the media file will continue after the transferring of the media file is complete by a period of time that equates to the buffering window. That is, in these embodiments, the time that the media file was transferred will equate to the time that the media file was displayed but will be shifted by the buffering window. To the extent Uniloc contends the plain and ordinary meaning of this term focuses on a play time, rather than a transfer time, of the "digital media presentation data," in my opinion *Hayward* teaches that the time stamps for the subsequent messages can be used to calculate "the approximate length of time that the embedded media player page was left open by the user." *Id.* ¶0063.

120. This understanding of *Hayward* is consistent with *Hayward*'s definition of "[s]treaming media files" as those "delivered over the Internet or other network environment to a client and playback on the client begins before the delivery of the entire file is completed." *Id.* ¶0022. This understanding of *Hayward* is also consistent with the example media file formats *Hayward* expressly contemplates, including "REALAUDIO^{TM}, REALVIDEO#, MICROSOFT WINDOWS MEDIA FORMAT^{TM}, FLASH^{TM}, [and] APPLE QUICKTIME^{TM}." *Id.* ¶0023.

121. Because the media file identification message in *Hayward* thus indicates when the media file begins to play in the embedded media player page, and each subsequent message in *Hayward* thus indicates that the embedded media player is still open during the predetermined interval, in my opinion a POSA would

have understood *Hayward* to disclose that "the stored data is indicative of an amount of time the digital media presentation data is streamed from the second computer system to the user's computer."

i. [1i] "wherein each stored data is together indicative of a cumulative time the corresponding web page was displayed by the user's computer"

122. In my opinion, *Hayward* discloses that each stored data (entries at the media file index and log system 122 relating to the media file identification message and the subsequent messages) is together indicative of a cumulative time the corresponding web page (embedded media player page) was displayed by the user's computer (client 110).

123. *Hayward* discloses that the media file index and log server 122 stores a "time stamp . . . identifying the time at which the media file identification message was transmitted," and "the time that the media file identification message was received" in a log associated with the media file. *Hayward*, ¶0058. Similarly, *Hayward* discloses that a time stamp for each subsequent message is "stored in the log associated with the media file." *Id.* ¶0060. *Hayward* explains that "by calculating the difference in time between the first and last time stamps for a media file during a selected playing session recorded in the log, the approximate length of time that the embedded media player page was left open by the user can be calculated." *Id.* ¶0063.

124. Because *Hayward* thus discloses that the media file identification message and the subsequent messages are indicative of how long the embedded media player page was open at client 110, in my opinion *Hayward* discloses that "each stored data is together indicative of a cumulative time the corresponding web page was displayed by the user's computer."

B. Ground 2: *Hayward* and *Middleton* render obvious claim 1

1. Claim 1

a. [1a] "A method for tracking digital media presentations delivered from a first computer system to a user's computer via a network comprising:"

125. In my opinion, *Hayward* discloses this element, as described in Section VIII.A.1.a.

b. [1b] "providing a corresponding web page to the user's computer for each digital media presentation to be delivered using the first computer system;"

126. In my opinion, *Hayward* discloses this element, as described in Section VIII.A.1.b.

c. [1c] "providing identifier data to the user's computer using the first computer system"

127. In my opinion, *Hayward* discloses this element, as described in Section VIII.A.1.a.

d. [1d] "providing an applet to the user's computer for each digital media presentation to be delivered using

the first computer system, wherein the applet is operative by the user's computer as a timer"

128. In my opinion, *Hayward* and *Middleton* would have rendered obvious to a POSA providing an applet (applet 44 in *Middleton*) to the user's computer (client 110) for each digital media presentation (media file) to be delivered using the first computer system (customer system 118 and media file index and log system 122), wherein the applet is operative by the user's computer (client 110) as a timer.

129. In *Hayward*, when the user clicks a link for a selected media file, "the customer system 122 instructs the client to request [an] embedded media player page from the customer system 122," and the media player displays the selected media file. *Hayward*, ¶¶0029, 0046.

130. According to *Hayward*, the embedded media player page also "facilitates the collection of data in connection with the playing of the video file." *Id.* ¶0057. In particular, "the embedded media player page instructs the client 110 to transmit a media file identification message to [the] media file index and log server 122." *Id.* ¶0058. Additionally, the embedded media player page "instruct[s] the client 110 to transmit at least one subsequent message . . . while the embedded media player page remains open." *Id.* ¶0060.

131. *Hayward* states that the embedded media player page instructs the client 110 to send the subsequent messages using "scripting." As discussed above

in Section VIII.A.1.d, it is my opinion that a POSA would have understood from *Hayward* that, as with the subsequent messages, the embedded media player page uses “scripting” to instruct the client 110 to send the media file identification message. *Hayward* defines “scripting,” as used in *Hayward*, as “server or client-side programming which supplements a static HTML page.” *Id.* ¶0017. *Hayward* contemplates that its “scripting” could be, for example, “Java” or “JavaScript.” *Id.*

132. Like the claimed applet, which is “operative by the user’s computer as a timer,” *Hayward*’s scripting is operative by the client to instruct the client to send the subsequent messages “at predetermined time intervals.” *Id.* ¶0060; *see also id.* ¶0061 (“periodic intervals,” “every thirty seconds”). From the media file identification message and the subsequent messages, *Hayward* teaches, “much information can be gleaned about the user and the playing event,” such as “the approximate length of time that the embedded media player page was left open.” *Id.* ¶0063.

133. As noted above in Section VIII.A.1.d, it is my opinion that a POSA would have understood that *Hayward* defines “scripting,” as used in *Hayward*, to include applets. But even if not, it would have been obvious to a POSA to use an applet to send the media file identification message and subsequent messages in *Hayward*, because it was well known at the time the provisional application leading to the ’609 patent was filed to use applets to track a user’s viewing of

content in a web page as evidenced by, for example, *Middleton*.

134. *Middleton* describes a “Web page” that includes an object, such as an advertisement. *Middleton*, ¶0028. *Middleton*’s web page includes “Java^{TM} code 44 that includes instructions to be run while [a] user computer 20a is displaying the web page.” *Id.* ¶0026. The Java^{TM} code 44 “includes an applet program and data for tracking and logging the activities of the user in memory 24 while the user is viewing the Web page.” *Id.* ¶0029. For example, the applet may permit an advertiser to track “the elapsed time that [an] element 48 [of the advertisement 39] has been displayed on the page.” *Id.* ¶0037. In this manner, “[t]he applet program 44 . . . permits the authors of the advertisement 39 to better understand how the users interact with the Web page advertisement.” *Id.* ¶0029.

135. In my opinion, a POSA would have been motivated and would have found it obvious to implement *Hayward*’s media file identification message and subsequent messages using an applet, as in *Middleton*, because (i) *Hayward* shows that JavaScript and a Java applet were known alternatives for adding a feature to a web page; (ii) *Hayward*’s scripting and *Middleton*’s applet add similar tracking features in similar web pages; and (iii) a POSA would have understood that an applet would have provided technical benefits to *Hayward*’s customer system 118 and media file index and log system 122.

136. A POSA would have known at the time the provisional application

leading to the ’609 patent was filed that JavaScript and a Java applet were identified and predictable alternatives for adding a feature to a web page. *See*, *e.g.*, Appendix C at 1623 (listing among the “MOST COMMONLY USED NETWORK PROGRAMMING TOOLS” Java and JavaScript; characterizing Java and JavaScript as “well-developed network programming tools available today.”). *Hayward* itself recognizes this. As *Hayward* explains, “programming which supplements a static HTML page” could be written either in JavaScript or as a Java applet. *Hayward*, ¶0017.

137. Moreover, a POSA would have recognized that *Hayward*’s scripting and *Middleton*’s applet add similar tracking features in similar web pages. Like *Hayward*’s embedded media player page, which is displayed in “a browser, such as Microsoft Internet Explorer, of a client,” and facilitates display of a media file, *id.* ¶¶0031–0032, *Middleton*’s web page is downloaded using “browser program software such as . . . Microsoft Internet Explorer^{TM}” and facilitates display of an advertisement that includes “graphics, pictures, or words,” *Middleton*, ¶¶0011, 0024. And, like *Hayward*’s scripting, which permits tracking of “the approximate length of time that the embedded media player page was left open,” *Hayward*, ¶0063, *Middleton*’s applet permits tracking of “the elapsed time that [an] element 48 [of the advertisement 39] has been displayed on the page,” *Middleton*, ¶0037. In my opinion, a POSA would have found it obvious to use an applet, like

Middleton's, to generate the media file identification messages and the subsequent messages to track the length of time the embedded media player page is open, just as the applet is used in *Middleton* to track how long the advertisement has been displayed.

138. In my opinion, a POSA would have understood that an applet would have provided technical benefits to *Hayward*'s customer system 118 and media file index and log system 122 at the time the provisional application leading to the '609 patent was filed. Based on my experience, a POSA would have known that one benefit was that Java applets allowed for the creation of "threads." *See, e.g.*, Appendix B at 253; Appendix C at 1625. JavaScript did not. *See, e.g.*, Appendix A at 255 ("The core JavaScript language does not contain any threading mechanism, and client-side JavaScript does not add any.") With threads, a programmer could separate an activity into multiple tasks that execute concurrently (e.g., in parallel) on different threads. Appendix B at 253. One advantage of using threads was that a short-running task (e.g., reacting to a user's mouse click) could be completed quickly while a long-running task (e.g., a file download) continued to make progress. *Id.* In my experience, Java achieved this concurrency with minimal programmer effort.

139. Based on my experience, a POSA would have known that another benefit was that Java applets provided class-based inheritance. *See, e.g.*,

Appendix B at 91–93. JavaScript did not. Appendix A at 157 (noting that JavaScript has prototype-based, rather than class-based, inheritance). A "class" in Java was source code that defined data values and methods. Appendix B at 58. Class-based inheritance meant that new classes could be created that inherited, or copied, the data values and methods defined in another class, thereby reducing the size of the class definition. *Id.* at 91–93. Class-based inheritance also allowed for "overriding," in which a programmer copied some, but not all, of the methods defined in another class. *Id.* at 94. In this manner, the programmer could take advantage of the reduced class definition while still tailoring a new class to a specific need. *Id.*

140. In my experience, these and other benefits of Java were appealing to programmers and provided significant advantages for large software systems like *Hayward*'s customer system 118 and media file index and log system 122. *See, e.g.*, Appendix C at 1630 ("Java can be used on the client side as well as on the server side. IT allows implementing a complex functionality of a larger program by using object oriented and well-structured language."). As one example, threading allowed for modularization, in which distinct threads were used to carry out distinct processes, even if the processes were to run concurrently. *See, e.g.*, Appendix B at 253. In my experience, such modularization could simplify the development and maintenance of large software systems like *Hayward*'s customer

system 118 and media file index and log system 122, because it allowed a discrete code element to be designed, implemented, and/or modified without affecting the myriad other aspects of the software.

141. Based on my experience, a POSA would have known that still another benefit was that Java applets were faster than some other technologies, including JavaScript. *See, e.g.*, Appendix C at 1625 (describing as a "feature" that "ensured success and increased importance" of Java that it is "fast"). JavaScript was an interpreted, rather than compiled language, meaning the source code itself was passed with a web page, and the browser at the client converted it to machine code upon receipt. *Id.* at 1624 ("JavaScript is an interpretive language and the scripts run as the Web page is downloaded and displayed."). As a result, JavaScript was inherently slower than Java applets.

142. Based on my experience, it is my opinion that a POSA would also have understood that an applet would have provided additional technical benefits to *Hayward*'s customer system 118 and media file index and log system 122 in embodiments where the media player is embedded using an applet. *Hayward* teaches that, in some embodiments, "a functional media player applet may accompany a data page download, and the data page is configured to embed the media player generated by the applet." *Hayward*, ¶0002. In these embodiments, a POSA would have been motivated to send the media file identification message

and the subsequent messages using an applet as well, because in my experience using consistent technologies between the media player and the subsequent messages would simplify the design and maintenance of the customer system 118 and the media file index and log system 122.

143. Because *Hayward* thus teaches providing scripting to the client 110 in the embedded media player page for each video file delivered using the customer system 118, and that the scripting sends messages at predetermined time intervals, and in my opinion a POSA would have found it obvious and would have been motivated to use an applet, as in *Middleton*, to send *Hayward*'s subsequent messages, in my opinion a POSA would have understood *Hayward* and *Middleton* to render obvious "providing an applet to the user's computer for each digital media presentation to be delivered using the first computer system, wherein the applet is operative by the user's computer as a timer."

e. [1e] "receiving at least a portion of the identifier data from the user's computer responsively to the timer applet each time a predetermined temporal period elapses using the first computer system"

144. In my opinion, *Hayward* and *Middleton* teach receiving at least a portion of the identifier data (the unique identifier for the media file in *Hayward*) from the user's computer (client 110 in *Hayward*) responsively to the timer applet (applet 44 in *Middleton*) each time a predetermined temporal time period elapses using the first computer system (customer system 118 and media file index and log

server 122 in *Hayward*).

145. In *Hayward*, the media file index and log system 122 receives subsequent messages sent using the scripting at the client 110 each time a predetermined temporal time period elapses. In my opinion, as discussed above in Section VIII.A.1.e, a POSA would have understood that the subsequent messages would have uniquely identified the media file. *Hayward*'s media file index and log system 122 indexes "a plurality of media files," each of which is identified by "a unique identifier for the media file," and "maintains a respective log for each indexed media file." *Hayward*, ¶¶0027, 0058. When a subsequent message is received in *Hayward*, "a time stamp for the . . . subsequent message is stored in the log associated with the media file." *Id.* ¶¶0060. In my opinion, a POSA would have understood that the subsequent messages uniquely identify the media file so that the media file index and log system 122 can store the time stamp in the "log associated with the media file."

146. In my opinion, the POSA would have understood *Hayward* to suggest that the subsequent message could uniquely identify the media file by including, for example, the unique identifier for the media file. *Hayward* teaches that each of the media file identification message and the subsequent messages takes the form of "an HTTP request to the media file index and log system 122 for a one-pixel GIF file." *Id.* ¶¶0059, 0061. And *Hayward* states that the unique identifier for the

media file is “appended to the HTTP request” that is the media file identification message. *Id.* ¶0059. In my opinion, a POSA would have understood *Hayward* to suggest that the HTTP request that is each subsequent message, like the HTTP request that is the media file identification message, similarly appends the unique identifier for the media file. This would have allowed the time stamps for the subsequent messages to be “stored in the log associated with the media file” by the media file index and log system 122, as *Hayward* describes. *Id.* at ¶0060.

147. In my opinion, a POSA would have found it obvious and would have been motivated to use an applet, as in *Middleton*, to send *Hayward*’s subsequent messages, as described in Section VIII.B.1.d.

148. Because in my opinion a POSA would have understood *Hayward* to teach the media file index and log system 122 receiving subsequent messages including the unique identifier for the media file sent using the scripting at the client 110 each time a predetermined temporal time period elapses, and in my opinion a POSA would have found it obvious and would have been motivated to use an applet, as in *Middleton*, to send *Hayward*’s subsequent messages, it is my opinion that a POSA would have understood *Hayward* and *Middleton* to teach “receiving at least a portion of the identifier data from the user's computer responsively to the timer applet each time a predetermined temporal period elapses using the first computer system.”

f. [1f] "storing data indicative of the received at least portion of the identifier data using the first computer system"

149. In my opinion, *Hayward* discloses this element, as described in Section VIII.A.1.f.

g. [1g] "wherein each provided webpage causes corresponding digital media presentation data to be streamed from a second computer system distinct from the first computer system directly to the user's computer independent of the first computer system"

150. In my opinion, *Hayward* discloses this element, as described in Section VIII.A.1.g.

h. [1h] "wherein the stored data is indicative of an amount of time the digital media presentation data is streamed from the second computer system to the user's computer"

151. In my opinion, *Hayward* discloses this element, as described in Section VIII.A.1.h.

i. [1i] "wherein each stored data is together indicative of a cumulative time the corresponding web page was displayed by the user's computer"

152. In my opinion, *Hayward* discloses this element, as described in Section VIII.A.1.i.

C. Ground 3: *Hayward*, or *Hayward* and *Middleton*, and *Ryan* render obvious claims 2 and 3

1. [2] "The method of claim 1, wherein the storing comprises incrementing a stored value dependently upon the receiving."

153. In my opinion, *Hayward*, as discussed in Ground 1, or *Hayward* and *Middleton*, as discussed in Ground 2, and *Ryan* teach that the storing (storing the time stamps for the subsequent messages in *Hayward*) comprises incrementing a stored value (weighting factor X in *Ryan*) dependently upon the receiving.

154. As discussed above in Section VIII.A.1.e (or Section VIII.B.1.e), in my opinion *Hayward* discloses (or *Hayward* and *Middleton* teach) an applet at the client 110 that sends subsequent messages to the media file index and log system 122. *Hayward* also teaches that a time stamp for each subsequent message is "stored in the log associated with the media file" at the media file index and log server 122. *Hayward*, ¶0060. "[B]y calculating the difference in time between the first and last time stamps for a media file during a selected playing session recorded in the log, the approximate length of time that the embedded media player page was left open by the user can be calculated." *Id.* ¶0063.

155. *Ryan* teaches that the weighting factor X may be "increment[ed] . . . based on the time spent at the web page," as determined from the surfer trace data indicating "the difference between two time date/time data 132 from subsequent selections from the list of web page searches." *Ryan*, 9:22–25, 16:40–41. In *Ryan*,

"[t]he longer the time spent" at the web page, "the more this increments the value of X." *Id.* at 16:41–42.

156. In my opinion, a POSA would have found it obvious to modify *Hayward*'s media file index and log server 122 to increment a stored value, like *Ryan*'s weighting factor X, dependently upon receiving the subsequent messages. *Hayward*'s (or *Hayward* and *Middleton*'s) applet at the client 110 sends subsequent messages to the media file index and log system 122. In my opinion, it would have been obvious to a POSA to modify the media file index and log system 122 to increment a stored value based on their receipt. The POSA would have known to do so based on *Ryan*, as *Ryan*'s server increments weighting factor X based on surfer trace data received from an applet at the user's personal computer. *Id.* at 8:63–67, 9:22–30, 9:41–56, 16:40–41.

157. In my opinion, *Ryan* itself would have motivated the POSA to modify *Hayward*'s media file index and log server 122 to increment a stored value, like *Ryan*'s weighting factor X. *Ryan* highlights the value to advertisers of knowing not only how long a user spent on a web page, as indicated by the "surfer trace data," i.e., "[t]he time difference between [] two selections" from a list of search results, but also how interesting a web page is to users, as indicated by the weighting factor X. *Id.* at 9:22–30, 12:22–30. *Ryan* shows that the weighting factor may provide a distinct value from the surfer trace data, insofar as it may

only reflect users showing sufficient interest in a web page. For instance, in *Ryan*, while surfer trace data may be collected for a web page view of any length, the weighting factor X may be incremented only when "the user exceed[s] a specified time at a location." *Id.* at 16:34–39.

158. Like *Ryan*, *Hayward* contemplates providing valuable data to advertisers, but the time stamps for the media file identification message and the subsequent messages stored in *Hayward*'s media file index and log system 122 only indicate, for example, how long the embedded media player page was displayed. *Hayward*, ¶0064. Based on *Ryan*, a POSA would have been motivated to modify *Hayward*'s media file index and log system 122 to also store data indicating how interesting the digital media presentation displayed by the embedded media player page is to users. In my opinion, looking to *Ryan*, the POSA would have been motivated to modify *Hayward*'s media file index and log system 122 to store a value akin to weighting factor X in *Ryan* and to increment the stored value dependently on receipt of the subsequent messages.

159. Because *Ryan* thus discloses incrementing the weighting factor X based on received surfer trace data indicating how long a user spent at a web page, and in my opinion a POSA would have been motivated and would have found it obvious to modify *Hayward*'s media file index and log server 122 to increment a stored value as in *Ryan* based on receipt of the subsequent messages, it is my

opinion that a POSA would have understood that the combination of *Hayward* (or *Hayward* and *Middleton*) and *Ryan* renders obvious that "the storing comprises incrementing a stored value dependently upon the receiving."

2. [3] "The method of claim 2, wherein the received data is indicative of a temporal cycle passing."

160. In my opinion, *Hayward* discloses that the received data (subsequent messages) is indicative of a temporal cycle ("predetermined interval") passing.

161. In *Hayward*, each subsequent message is sent "at [a] predetermined time interval[],"such as "every thirty seconds." *Hayward*, ¶¶0060–0061. A time stamp for each subsequent message is "stored in the log associated with the media file." *Id.* ¶0060.

162. Because *Hayward* thus teaches that the subsequent messages are sent at predetermined time intervals and stored with time stamps by the media file index and log system 122, in my opinion *Hayward* discloses that "the received data is indicative of a temporal cycle passing."

IX. Conclusion

163. As discussed above, it is my opinion that the Challenged Claims of the '609 patent are not patentable.

164. In signing this declaration, I recognize that the declaration will be filed as evidence in a contested proceeding before the Patent Trial and Appeal Board of the United States Patent and Trademark Office. I also recognize that I

may be subject to cross-examination in the proceeding and that cross-examination will take place within the United States. If cross-examination is required of me, I will appear for cross-examination within the United States during the time allotted for cross-examination.

165. I reserve the right to supplement my opinions in the future to respond to any arguments that Patent Owner raises and to account for new information as it becomes available to me.

166. I declare that all statements made herein of my own knowledge are true and that all statements made on information and belief are believed to be true; and further that these statements were made with the knowledge that willful false statements and the like so made are punishable by fine or imprisonment, or both, under Section 1001 of Title 18 of the United States Code.

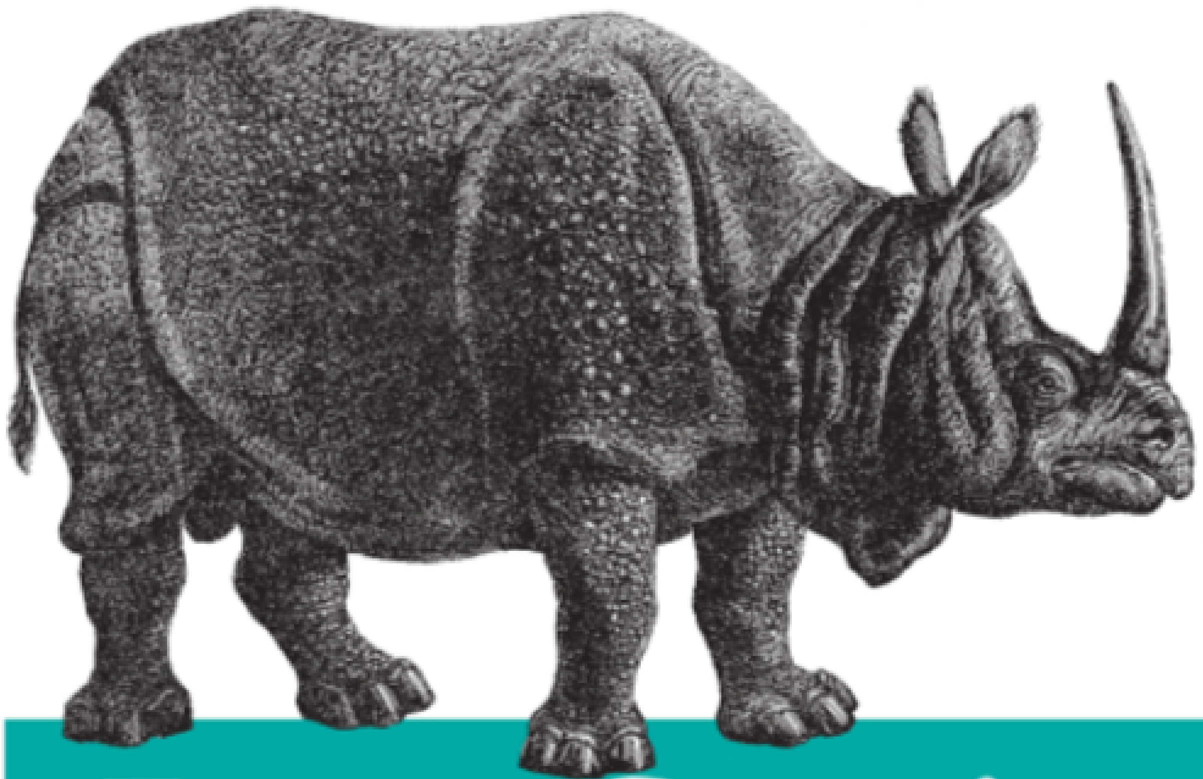
Executed on this 31st day of October 2019, in Durham, North Carolina.

By: [illegible]
Dr. Jeffrey S. Chase

APPENDIX A

Activate Your Web Pages

5th Edition
Covers Ajax and DOM Scripting



JavaScript

The Definitive Guide

O'REILLY®

David Flanagan

FIFTH EDITION

JavaScript

The Definitive Guide

David Flanagan

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Sebastopol • Taipei • Tokyo

JavaScript: The Definitive Guide, Fifth Edition
by David Flanagan

Copyright © 2006, 2002, 1998, 1997, 1996 O'Reilly Media, Inc. All rights reserved.
Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*safari.oreilly.com*). For more information, contact our corporate/institutional sales department: (800) 998-9938 or *corporate@oreilly.com*.

Editor: Debra Cameron
Production Editor: Sanders Kleinfeld
Copyeditor: Mary Anne Weeks Mayo
Proofreader: Sanders Kleinfeld
Indexer: Ellen Troutman-Zaig
Cover Designer: Edie Freedman
Interior Designer: David Futato
Illustrators: Robert Romano and Jessamyn Read

Printing History:

August 1996:	Beta Edition.
January 1997:	Second Edition.
June 1998:	Third Edition.
January 2002:	Fourth Edition.
August 2006:	Fifth Edition.

Nutshell Handbook, the Nutshell Handbook logo, and the O'Reilly logo are registered trademarks of O'Reilly Media, Inc. *JavaScript: The Definitive Guide*, the image of a Javan rhinoceros, and related trade dress are trademarks of O'Reilly Media, Inc. Java™, all Java-based trademarks and logos, and JavaScript™ are trademarks or registered trademarks of Sun Microsystems, Inc., in the United States and other countries. O'Reilly Media, Inc. is independent of Sun Microsystems.

Mozilla and Firefox are registered trademarks of the Mozilla Foundation. Netscape and Netscape Navigator are registered trademarks of America Online, Inc. Internet Explorer and the Internet Explorer Logo are trademarks and tradenames of Microsoft Corporation. All other product names and logos are trademarks of their respective owners.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and O'Reilly Media, Inc. was aware of a trademark claim, the designations have been printed in caps or initial caps.

While every precaution has been taken in the preparation of this book, the publisher and author assume no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.

ISBN: 978-0-596-10199-2
[M] [5/09]

CHAPTER 9

Classes, Constructors, and Prototypes

JavaScript objects were introduced in Chapter 7. That chapter treated each object as a unique set of properties, different from every other object. In many object-oriented programming languages, it is possible to define a *class* of objects and then create individual objects that are *instances* of that class. You might define a class named Complex to represent and perform arithmetic on complex numbers, for example. A Complex object would represent a single complex number and would be an instance of that class.

JavaScript does not support true classes the way that languages like Java, C++, and C# do.* Still, however, it is possible to define pseudoclasses in JavaScript. The tools for doing this are constructor functions and prototype objects. This chapter explains constructors and prototypes and includes examples of several JavaScript pseudoclasses and even pseudosubclasses.

For lack of a better term, I use the word "class" informally in this chapter. Be careful, however, that you don't confuse these informal classes with the true classes of JavaScript 2 and other languages.

9.1 Constructors

Chapter 7 showed you how to create a new, empty object either with the object literal {} or with the following expression:

```
new Object()
```

You have also seen other kinds of JavaScript objects created with a similar syntax:

```
var array = new Array(10);
var today = new Date();
```

The new operator must be followed by a function invocation. It creates a new object, with no properties and then invokes the function, passing the new object as the value of the this keyword. A function designed to be used with the new operator is called a

* True classes are planned for JavaScript 2.0, however.

constructor function or simply a constructor. A constructor's job is to initialize a newly created object, setting any properties that need to be set before the object is used. You can define your own constructor function, simply by writing a function that adds properties to this. The following code defines a constructor and then invokes it twice with the new operator to create two new objects:

```
// Define the constructor.
// Note how it initializes the object referred to by "this".
function Rectangle(w, h) {
    this.width = w;
    this.height = h;
    // Note: no return statement here
}

// Invoke the constructor to create two Rectangle objects.
// We pass the width and height to the constructor
// so that it can initialize each new object appropriately.
var rect1 = new Rectangle(2, 4);    // rect1 = { width:2, height:4 };
var rect2 = new Rectangle(8.5, 11); // rect2 = { width:8.5, height:11 };
```

Notice how the constructor uses its arguments to initialize properties of the object referred to by the this keyword. You have defined a class of objects simply by defining an appropriate constructor function; all objects created with the Rectangle() constructor are now guaranteed to have initialized width and height properties. This means that you can write programs that rely on this fact and treat all Rectangle objects uniformly. Because every constructor defines a class of objects, it is stylistically important to give a constructor function a name that indicates the class of objects it creates. Creating a rectangle with new Rectangle(1,2) is a lot more intuitive than with new init_rect(1,2), for example.

Constructor functions typically do not have return values. They initialize the object passed as the value of this and return nothing. However, a constructor is allowed to return an object value, and, if it does so, that returned object becomes the value of the new expression. In this case, the object that was the value of this is simply discarded.

9.2 Prototypes and Inheritance

Recall from Chapter 8 that a *method* is a function that is invoked as a property of an object. When a function is invoked in this way, the object through which it is accessed becomes the value of the this keyword. Suppose you'd like to compute the area of the rectangle represented by a Rectangle object. Here is one way to do it:

```
function computeAreaOfRectangle(r) { return r.width * r.height; }
```

This works, but it is not object-oriented. When using objects, it is better to invoke a method on the object rather than passing the object to a function. Here's how to do that:

```
// Create a Rectangle object
var r = new Rectangle(8.5, 11);
// Add a method to it
r.area = function() { return this.width * this.height; }
```

```
// Now invoke the method to compute the area
var a = r.area();
```

Having to add a method to an object before you can invoke it is silly, of course. You can improve the situation by initializing the area property to refer to the area computing function in the constructor. Here is an improved `Rectangle()` constructor:

```
function Rectangle(w, h) {
    this.width = w;
    this.height = h;
    this.area = function() { return this.width * this.height; }
}
```

With this new version of the constructor, you can write code like this:

```
// How big is a sheet of U.S. Letter paper in square inches?
var r = new Rectangle(8.5, 11);
var a = r.area();
```

This solution works better but is still not optimal. Every rectangle created will have three properties. The `width` and `height` properties may be different for each rectangle, but the `area` of every single Rectangle object always refers to the same function (someone might change it, of course, but you usually intend the methods of an object to be constant). It is inefficient to use regular properties for methods that are intended to be shared by all objects of the same class (that is, all objects created with the same constructor).

There is a solution, however. It turns out that every JavaScript object includes an internal reference to another object, known as its prototype object. Any properties of the prototype appear to be properties of an object for which it is the prototype. Another way of saying this is that a JavaScript object *inherits* properties from its prototype.

In the previous section, I showed that the `new` operator creates a new, empty object and then invokes a constructor function as a method of that object. This is not the complete story, however. After creating the empty object, `new` sets the prototype of that object. The prototype of an object is the value of the `prototype` property of its constructor function. All functions have a `prototype` property that is automatically created and initialized when the function is defined. The initial value of the `prototype` property is an object with a single property. This property is named `constructor` and refers back to the constructor function with which the prototype is associated. (You may recall the `constructor` property from Chapter 7; this is why every object has a `constructor` property.) Any properties you add to this prototype object will appear to be properties of objects initialized by the constructor.

This is clearer with an example. Here again is the `Rectangle()` constructor:

```
// The constructor function initializes those properties that
// will be different for each instance.
function Rectangle(w, h) {
    this.width = w;
    this.height = h;
}
```

```
// The prototype object holds methods and other properties that
// should be shared by each instance.
Rectangle.prototype.area = function() { return this.width * this.height; }
```

A constructor provides a name for a "class" of objects and initializes properties, such as `width` and `height`, that may be different for each instance of the class. The prototype object is associated with the constructor, and each object initialized by the constructor inherits exactly the same set of properties from the prototype. This means that the prototype object is an ideal place for methods and other constant properties.

It can also be useful to use prototype-based inheritance without constructors and classes. The following simple function creates a new object with a specified prototype. That is, it creates a new object that inherits from, or is an "heir" of its argument*:

```
// Create and return an object that has p as its prototype
function heir(p) {
    function f(){}    // A dummy constructor function
    f.prototype = p;  // Specify the prototype object we want
    return new f();   // Invoke the constructor to create new object
}
```

Note that inheritance occurs automatically as part of the process of looking up a property value. Properties are *not* copied from the prototype object into new objects; they merely appear as if they were properties of those objects. This has two important implications. First, the use of prototype objects can dramatically decrease the amount of memory required by each object because the object can inherit many of its properties. The second implication is that an object inherits properties even if they are added to its prototype *after* the object is created. This means that it is possible (though not necessarily a good idea) to add new methods to existing classes.

Inherited properties behave just like regular properties of an object. They are enumerated by the `for/in` loop and can be tested with the `in` operator. You can distinguish them only with the `Object.hasOwnProperty()` method:

```
var r = new Rectangle(2, 3);
r.hasOwnProperty("width");   // true: width is a direct property of r
r.hasOwnProperty("area");    // false: area is an inherited property of r
"area" in r;                 // true: "area" is a property of r
```

9.2.1 Reading and Writing Inherited Properties

Each class has one prototype object, with one set of properties. But there are potentially many instances of a class, each of which inherits those prototype properties. Because one prototype property can be inherited by many objects, JavaScript must enforce a fundamental asymmetry between reading and writing property values. When you read property `p` of an object `o`, JavaScript first checks to see if `o` has a property named `p`. If it does not, it next checks to see if the prototype object of `o` has a property named `p`. This is what makes prototype-based inheritance work.

* Douglas Crockford calls this function <literal>Object.create()</literal> and describes it at *http://javascript.crockford.com/prototypal.html*

When you write the value of a property, on the other hand, JavaScript does not use the prototype object. To see why, consider what would happen if it did: suppose you try to set the value of the property `o.p` when the object `o` does not have a property named `p`. Further suppose that JavaScript goes ahead and looks up the property `p` in the prototype object of `o` and allows you to set the property of the prototype. Now you have changed the value of `p` for a whole class of objects—not at all what you intended.

Therefore, property inheritance occurs only when you read property values, not when you write them. If you set the property `p` in an object `o` that inherits that property from its prototype, what happens is that you create a new property `p` directly in `o`. Now that `o` has its own property named `p`, it no longer inherits the value of `p` from its prototype. When you read the value of `p`, JavaScript first looks at the properties of `o`. Since it finds `p` defined in `o`, it doesn't need to search the prototype object and never finds the value of `p` defined there. We sometimes say that the property `p` in `o` "shadows" or "hides" the property `p` in the prototype object. Prototype inheritance can be a confusing topic. Figure 9-1 illustrates the concepts discussed here.

Because prototype properties are shared by all objects of a class, it generally makes sense to use them to define only properties that are the same for all objects within the class. This makes prototypes ideal for defining methods. Other properties with constant values (such as mathematical constants) are also suitable for definition with prototype properties. If your class defines a property with a very commonly used default value, you might define this property and its default value in a prototype object. Then, the few objects that want to deviate from the default value can create their own private, unshared copies of the property and define their own nondefault values.

9.2.2 Extending Built-in Types

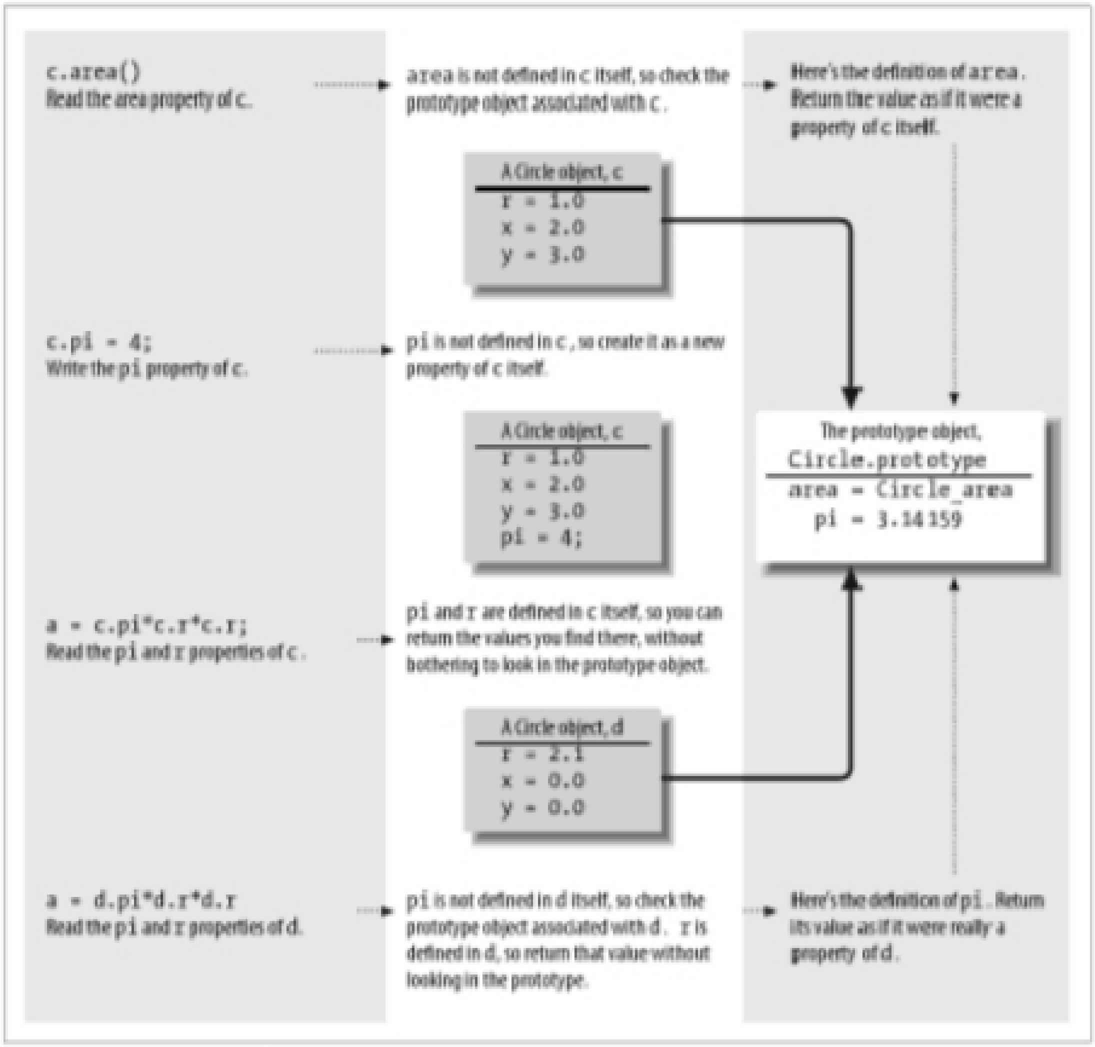
It is not only user-defined classes that have prototype objects. Built-in classes, such as String and Date, have prototype objects too, and you can assign values to them. For example, the following code defines a new method that is available for all String objects:

```
// Returns true if the last character is c
String.prototype.endsWith = function(c) {
    return (c == this.charAt(this.length-1))
}
```

Having defined the new `endsWith()` method in the String prototype object, you can use it like this:

```
var message = "hello world";
message.endsWith('h')  // Returns false
message.endsWith('d')  // Returns true
```

There is a strong argument against extending built-in types with your own methods; if you do so, you are essentially creating your own custom version of the core JavaScript API. Any other programmers who have to read or maintain your code will likely find it confusing if your code includes methods they have never heard of.



c.area()
Read the area property of c.
area is not defined in c itself, so check the prototype object associated with c.
Here's the definition of area. Return the value as if it were a property of c itself.
A Circle object, c
r = 1.0
x = 2.0
y = 3.0
c.pi = 4;
Write the pi property of c.
pi is not defined in c, so create it as a new property of c itself.
A Circle object, c
r = 1.0
x = 2.0
y = 3.0
pi = 4;
The prototype object, Circle.prototype
area = Circle_area
pi = 3.14159
a = c.pi*c.r*c.r;
Read the pi and r properties of c.
pi and r are defined in c itself, so you can return the values you find there, without bothering to look in the prototype object.
A Circle object, d
r = 2.1
x = 0.0
y = 0.0
a = d.pi*d.r*d.r
Read the pi and r properties of d.
pi is not defined in d itself, so check the prototype object associated with d. r is defined in d, so return that value without looking in the prototype.
Here's the definition of pi. Return its value as if it were really a property of d.

Figure 9-1. Objects and prototypes

Unless you are creating a low-level JavaScript framework that you expect to be adopted by many other programmers, it is probably best to stay away from the prototype objects of the built-in types.

Note that you must *never* add properties to `Object.prototype`. Any properties or methods you add are enumerable with a `for/in` loop, and adding them to `Object.prototype` makes them visible in every single JavaScript object. An empty object, {}, is expected to have no enumerable properties. Anything added to `Object.prototype` becomes an enumerable property of the empty object and will likely break code that uses objects as associative arrays.

The technique shown here for extending built-in object types is guaranteed to work only for core JavaScript "native objects." When JavaScript is embedded in some context, such as a web browser or a Java application, it has access to additional "host objects" such as objects that represent web browser document content. These host objects do not typically have constructors and prototype objects, and you usually cannot extend them.

There is one case in which it is safe and useful to extend the prototype of a built-in native class: to add standard methods to a prototype when an old or incompatible JavaScript implementation lacks them. For example, the `Function.apply()` method is missing in Microsoft Internet Explorer 4 and 5. This is a pretty important function, and you may see code like this to replace it:

```
// IE 4 & 5 don't implement Function.apply().
// This workaround is based on code by Aaron Boodman.
if (!Function.prototype.apply) {
    // Invoke this function as a method of the specified object,
    // passing the specified parameters.  We have to use eval() to do this
    Function.prototype.apply = function(object, parameters) {
        var f = this;                // The function to invoke
        var o = object || window;    // The object to invoke it on
        var args = parameters || []; // The arguments to pass

        // Temporarily make the function into a method of o
        // To do this we use a property name that is unlikely to exist
        o._$_apply_$_ = f;

        // We will use eval() to invoke the method. To do this we've got
        // to write the invocation as a string. First build the argument list.
        var stringArgs = [];
        for(var i = 0; i < args.length; i++)
            stringArgs[i] = "args[" + i + "]";

        // Concatenate the argument strings into a comma-separated list.
        var arglist = stringArgs.join(",");

        // Now build the entire method call string
        var methodcall = "o._$_apply_$_(" + arglist + ");";

        // Use the eval() function to make the methodcall
        var result = eval(methodcall);

        // Unbind the function from the object
        delete o._$_apply_$_;

        // And return the result
        return result;
    };
}
```

As another example, consider the new array methods implemented in Firefox 1.5 (see Section 7.7.10). If you want to use the new `Array.map()` method but also want your code to work on platforms that do not support this method natively, you can use this code for compatibility:

```
// Array.map() invokes a function f on each element of the array,
// returning a new array of the values that result from each function
// call. If map() is called with two arguments, the function f
// is invoked as a method of the second argument. When invoked, f()
// is passed 3 arguments. The first is the value of the array
// element. The second is the index of the array element, and the
// third is the array itself. In most cases it needs to use only the
// first argument.
if (!Array.prototype.map) {
    Array.prototype.map = function(f, thisObject) {
        var results = [];
        for(var len = this.length, i = 0; i < len; i++) {
            results.push(f.call(thisObject, this[i], i, this));
        }
        return results;
    }
}
```

9.3 Simulating Classes in JavaScript

Although JavaScript supports a datatype called an object, it does not have a formal notion of a class. This makes it quite different from classic object-oriented languages such as C++ and Java. The common conception about object-oriented programming languages is that they are strongly typed and support class-based inheritance. By these criteria, it is easy to dismiss JavaScript as not being a true object-oriented language. On the other hand, you've seen that JavaScript makes heavy use of objects, and it has its own type of prototype-based inheritance. JavaScript is a true object-oriented language. It draws inspiration from a number of other (relatively obscure) object-oriented languages that use prototype-based inheritance instead of class-based inheritance.

Although JavaScript is not a class-based object-oriented language, it does a good job of simulating the features of class-based languages such as Java and C++. I've been using the term *class* informally throughout this chapter. This section more formally explores the parallels between JavaScript and true class-based inheritance languages such as Java and C++.*

Let's start by defining some basic terminology. An *object*, as you've already seen, is a data structure that contains various pieces of named data and may also contain various methods to operate on those pieces of data. An object groups related values and methods into a single convenient package, which generally makes programming easier by increasing the modularity and reusability of code. Objects in JavaScript may have any number of properties, and properties may be dynamically added to an object. This is not the case in strictly typed languages such as Java and C++. In those

* You should read this section even if you are not familiar with those languages and that style of object-oriented programming.

languages, each object has a predefined set of properties,* where each property is of a predefined type. When you use JavaScript objects to simulate class-based programming techniques, you generally define in advance the set of properties for each object and the type of data that each property holds.

In Java and C++, a *class* defines the structure of an object. The class specifies exactly what fields an object contains and what types of data each holds. It also defines the methods that operate on an object. JavaScript does not have a formal notion of a class, but, as shown earlier, it approximates classes with its constructors and their prototype objects.

In both JavaScript and class-based object-oriented languages, there may be multiple objects of the same class. We often say that an object is an *instance* of its class. Thus, there may be many instances of any class. Sometimes the term *instantiate* is used to describe the process of creating an object (i.e., an instance of a class).

In Java, it is a common programming convention to name classes with an initial capital letter and to name objects with lowercase letters. This convention helps keep classes and objects distinct from each other in code, and it is useful to follow in JavaScript programming as well. Previous sections of this chapter, for example, have defined a Rectangle class and created instances of that class with names such as `rect`.

The members of a Java class may be of four basic types: instance properties, instance methods, class properties, and class methods. In the following sections, we'll explore the differences between these types and show how they are simulated in JavaScript.

9.3.1 Instance Properties

Every object has its own separate copies of its *instance properties*. In other words, if there are 10 objects of a given class, there are 10 copies of each instance property. In our Rectangle class, for example, every Rectangle object has a property `width` that specifies the width of the rectangle. In this case, `width` is an instance property. Since each object has its own copy of the instance properties, these properties are accessed through individual objects. If `r` is an object that is an instance of the Rectangle class, for example, its width is referred to as:

```
r.width
```

By default, any object property in JavaScript is an instance property. To truly simulate traditional class-based object-oriented programming, however, we will say that instance properties in JavaScript are those properties that are created and initialized by the constructor function.

* They are usually called "fields" in Java and C++, but I'll refer to them as properties here since that is the JavaScript terminology.

9.3.2 Instance Methods

An *instance method* is much like an instance property, except that it is a method rather than a data value. (In Java, functions and methods are not data, as they are in JavaScript, so this distinction is more clear.) Instance methods are invoked on a particular object, or instance. The area( ) method of our Rectangle class is an instance method. It is invoked on a Rectangle object r like this:

```
a = r.area();
```

The implementation of an instance method uses the this keyword to refer to the object or instance on which it is invoked. An instance method can be invoked for any instance of a class, but this does not mean that each object contains its own private copy of the method, as it does with instance properties. Instead, each instance method is shared by all instances of a class. In JavaScript, an instance method for a class is defined by setting a property in the constructor's prototype object to a function value. This way, all objects created by that constructor share an inherited reference to the function and can invoke it using the method-invocation syntax shown earlier.

9.3.2.1 Instance methods and this

If you are a Java or C++ programmer, you may have noticed an important difference between instance methods in those languages and instance methods in JavaScript. In Java and C++, the scope of instance methods includes the this object. The body of an area method in Java, for example might simply be:

```
return width * height;
```

In JavaScript, however, you've seen that you must explicitly specify the this keyword for these properties:

```
return this.width * this.height;
```

If you find it awkward to have to prefix each instance field with this, you can use the with statement (covered in Section 6.18) in each of your methods. For example:

```
Rectangle.prototype.area = function() {
    with(this) {
        return width*height;
    }
}
```

9.3.3 Class Properties

A *class property* in Java is a property that is associated with a class itself, rather than with each instance of a class. No matter how many instances of the class are created, there is only one copy of each class property. Just as instance properties are accessed through an instance of a class, class properties are accessed through the class itself. Number.MAX_VALUE is an example of a class property in JavaScript: the MAX_VALUE property is accessed through the Number class. Because there is only one copy of each class property, class properties are essentially global. What is nice about them, however, is that they are associated with a class, and they have a logical niche—a position in the JavaScript namespace where they are not likely to be overwritten by other properties with the same name. As is probably clear, you simulate a class property in JavaScript simply by defining a property of the constructor function itself. For example, to create a class property `Rectangle.UNIT` to store a special 1x1 rectangle, you can do the following:

```
Rectangle.UNIT = new Rectangle(1,1);
```

`Rectangle` is a constructor function, but because JavaScript functions are objects, you can create properties of a function just as you can create properties of any other object.

9.3.4 Class Methods

A *class method* is associated with a class rather than with an instance of a class. Class methods are invoked through the class itself, not through a particular instance of the class. The `Date.parse()` method (which you can look up in Part III) is a class method. You always invoke it through the Date constructor object rather than through a particular instance of the Date class.

Because class methods are invoked through a constructor function, the `this` keyword does not refer to any particular instance of the class. Instead, it refers to the constructor function itself. (Typically, a class method does not use `this` at all.)

Like class properties, class methods are global. Because they do not operate on a particular object, class methods are generally more easily thought of as functions that happen to be invoked through a class. Again, associating these functions with a class gives them a convenient niche in the JavaScript namespace and prevents namespace collisions. To define a class method in JavaScript, simply make the appropriate function a property of the constructor.

9.3.5 Example: A Circle Class

The code in Example 9-1 is a constructor function and prototype object for creating objects that represent circles. It contains examples of instance properties, instance methods, class properties, and class methods.

Example 9-1. A circle class

```
// We begin with the constructor
function Circle(radius) {
    // r is an instance property, defined and initialized in the constructor.
    this.r = radius;
}
```

Example 9-1. A circle class (continued)

```
// Circle.PI is a class property--it is a property of the constructor function.
Circle.PI = 3.14159;

// Here is an instance method that computes a circle's area.
Circle.prototype.area = function() { return Circle.PI * this.r * this.r; }

// This class method takes two Circle objects and returns the
// one that has the larger radius.
Circle.max = function(a,b) {
    if (a.r > b.r) return a;
    else return b;
}

// Here is some code that uses each of these fields:
var c = new Circle(1.0);      // Create an instance of the Circle class
c.r = 2.2;                    // Set the r instance property
var a = c.area();             // Invoke the area() instance method
var x = Math.exp(Circle.PI);  // Use the PI class property in our own computation
var d = new Circle(1.2);      // Create another Circle instance
var bigger = Circle.max(c,d); // Use the max() class method
```

9.3.6 Example: Complex Numbers

Example 9-2 is another example, somewhat more formal than the last, that defines a class of objects in JavaScript. The code and the comments are worth careful study.

Example 9-2. A complex number class

```
/*
 * Complex.js:
 * This file defines a Complex class to represent complex numbers.
 * Recall that a complex number is the sum of a real number and an
 * imaginary number and that the imaginary number i is the
 * square root of -1.
 */

/*
 * The first step in defining a class is defining the constructor
 * function of the class. This constructor should initialize any
 * instance properties of the object. These are the essential
 * "state variables" that make each instance of the class different.
 */
function Complex(real, imaginary) {
    this.x = real;       // The real part of the number
    this.y = imaginary;  // The imaginary part of the number
}

/*
 * The second step in defining a class is defining its instance
 * methods (and possibly other properties) in the prototype object
 * of the constructor. Any properties defined in this object will
 * be inherited by all instances of the class. Note that instance
```

Example 9-2. A complex number class (continued)

```
 * methods operate on the this keyword. For many methods,
 * no other arguments are needed.
 */

// Return the magnitude of a complex number. This is defined
// as its distance from the origin (0,0) of the complex plane.
Complex.prototype.magnitude = function() {
    return Math.sqrt(this.x*this.x + this.y*this.y);
};

// Return a complex number that is the negative of this one.
Complex.prototype.negative = function() {
    return new Complex(-this.x, -this.y);
};

// Add a complex number to this one and return the sum in a new object.
Complex.prototype.add = function(that) {
    return new Complex(this.x + that.x, this.y + that.y);
}

// Multiply this complex number by another and return the product as a
// new Complex object.
Complex.prototype.multiply = function(that) {
    return new Complex(this.x * that.x - this.y * that.y,
                       this.x * that.y + this.y * that.x);
}

// Convert a Complex object to a string in a useful way.
// This is invoked when a Complex object is used as a string.
Complex.prototype.toString = function() {
    return "{" + this.x + "," + this.y + "}";
};

// Test whether this Complex object has the same value as another.
Complex.prototype.equals = function(that) {
    return this.x == that.x && this.y == that.y;
}

// Return the real portion of a complex number. This function
// is invoked when a Complex object is treated as a primitive value.
Complex.prototype.valueOf = function() { return this.x; }

/*
 * The third step in defining a class is to define class methods,
 * constants, and any needed class properties as properties of the
 * constructor function itself (instead of as properties of the
 * prototype object of the constructor). Note that class methods
 * do not use the this keyword: they operate only on their arguments.
 */
// Add two complex numbers and return the result.
// Contrast this with the instance method add()
Complex.sum = function (a, b) {
```

Example 9-2. A complex number class (continued)

```
    return new Complex(a.x + b.x, a.y + b.y);
};

// Multiply two complex numbers and return the product.
// Contrast with the instance method multiply()
Complex.product = function(a, b) {
    return new Complex(a.x * b.x - a.y * b.y,
                       a.x * b.y + a.y * b.x);
};

// Here are some useful predefined complex numbers.
// They are defined as class properties, and their names are in uppercase
// to indicate that they are intended to be constants (although it is not
// possible to make JavaScript properties read-only).
Complex.ZERO = new Complex(0,0);
Complex.ONE = new Complex(1,0);
Complex.I = new Complex(0,1);
```

9.3.7 Private Members

A common feature of traditional object-oriented languages such as Java and C++ is that the properties of a class can be declared private so that they are available only to the methods of the class and cannot be manipulated by code outside of the class. A common programming technique called *data encapsulation* makes properties private and allows read and write access to them only through special accessor methods. JavaScript can simulate this using closures (an advanced topic, covered in Section 8.8), but to do so, the accessor methods must be stored on each object instance; they cannot be inherited from the prototype object.

The following code illustrates how this is done. It implements an immutable Rectangle object whose width and height can never be changed and are available only through accessor methods:

```
function ImmutableRectangle(w, h) {
    // This constructor does not store the width and height properties
    // in the object it initializes.  Instead, it simply defines
    // accessor methods in the object.  These methods are closures and
    // the width and height values are captured in their scope chains.
    this.getWidth = function() { return w; }
    this.getHeight = function() { return h; }
}

// Note that the class can have regular methods in the prototype object.
ImmutableRectangle.prototype.area = function() {
    return this.getWidth() * this.getHeight();
};
```

Douglas Crockford is generally credited as the first person to discover (or at least to publish) this technique for defining private properties. His original discussion is at *http://www.crockford.com/javascript/private.html*.

9.4 Common Object Methods

When defining a new JavaScript class, there are several methods that you should always consider defining. These methods are detailed in the subsections that follow.

9.4.1 The toString( ) Method

The idea behind `toString()` is that each class of objects has its own particular string representation, so it should define an appropriate `toString()` method to convert objects to that string form. When you define a class, you should define a `toString()` method for it so that instances of the class can be converted to meaningful strings. The string should contain information about the object being converted because this is useful for debugging purposes. If the string representation is chosen carefully, it can also be useful in programs themselves. Additionally, you might consider adding a static `parse()` method to your class to parse a string output by `toString()` back into object form.

The Complex class of Example 9-2 includes a `toString()` method, and the following code shows a `toString()` method you can define for a Circle class:

```
Circle.prototype.toString = function () {
    return "[Circle of radius " + this.r + ", centered at ("
        + this.x + ", " + this.y + ").]";
}
```

With this `toString()` method defined, a typical Circle object might be converted to the string "[Circle of radius 1, centered at (0, 0).]".

9.4.2 The valueOf( ) Method

The `valueOf()` method is much like the `toString()` method, but it is called when JavaScript needs to convert an object to some primitive type other than a string—typically, a number. Where possible, the function should return a primitive value that somehow represents the value of the object referred to by the `this` keyword.

By definition, objects are not primitive values, so most objects do not have a primitive equivalent. Thus, the default `valueOf()` method defined by the Object class performs no conversion and simply returns the object on which it is invoked. Classes such as Number and Boolean have obvious primitive equivalents, so they override the `valueOf()` method to return appropriate primitive values. This is why Number and Boolean objects can behave so much like their equivalent primitive values.

Occasionally, you may define a class that has some reasonable primitive equivalent. In this case, you may want to define a custom `valueOf()` method for the class. In the Complex class of Example 9-2, you'll see that a `valueOf()` method was defined that returned the real part of the complex number. Thus, when a Complex object is used in a numeric context, it behaves as if it were a real number without its imaginary component. For example, consider the following code:

```
var a = new Complex(5,4);
var b = new Complex(2,1);
var c = Complex.sum(a,b);  // c is the complex number {7,5}
var d = a + b;             // d is the number 7
```

One note of caution about defining a `valueOf()` method: the `valueOf()` method can, in some circumstances, take priority over the `toString()` method when converting an object to a string. Thus, when you define a `valueOf()` method for a class, you may need to be more explicit about calling the `toString()` method when you want to force an object of that class to be converted to a string. To continue with the Complex example:

```
alert("c = " + c);              // Uses valueOf(); displays "c = 7"
alert("c = " + c.toString());   // Displays "c = {7,5}"
```

9.4.3 Comparison Methods

JavaScript equality operators compare objects by reference, not by value. That is, given two object references, they look to see if both references are to the same object. They do not check to see if two different objects have the same property names and values. It is often useful to be able to compare two objects for equality or even for relative order (as the < and > operators do). If you define a class and want to be able to compare instances of that class, you should define appropriate methods to perform those comparisons.

The Java programming language uses methods for object comparison, and adopting the Java conventions is a common and useful thing to do in JavaScript. To enable instances of your class to be tested for equality, define an instance method named `equals()`. It should take a single argument and return `true` if that argument is equal to the object it is invoked on. Of course it is up to you to decide what "equal" means in the context of your own class. Typically, you simply compare the instance properties of the two objects to ensure that they have the same values. The Complex class in Example 9-2 has an `equals()` method of this sort.

It is sometimes useful to compare objects according to some ordering. That is, for some classes, it is possible to say that one instance is "less than" or "greater than" another instance. You might order Complex numbers based on their `magnitude()`, for example. On the other hand, it is not clear that there is a meaningful ordering of Circle objects: do you compare them based on radius, X coordinate and Y coordinate, or some combination of these?

If you try to use objects with JavaScript's relation operators such as < and <=, JavaScript first calls the `valueOf()` method of the objects and, if this method returns a primitive value, compares those values. Since our Complex class has a `valueOf()` method that returns the real part of a complex number, instances of the Complex class can be compared as if they were real numbers with no imaginary part. This may or may not be what you actually want. To compare objects according to an explicitly defined ordering of your own choosing, you can (again, following Java convention) define a method named `compareTo()`.

The compareTo() method should accept a single argument and compare it to the object on which the method is invoked. If the this object is less than the argument object, compareTo() should return a value less that zero. If the this object is greater than the argument object, the method should return a value greater than zero. And if the two objects are equal, the method should return zero. These conventions about the return value are important, and they allow you to substitute the following expressions for relational and equality operators:

Replace this	With this
`a < b`	`a.compareTo(b) < 0`
`a <= b`	`a.compareTo(b) <= 0`
`a > b`	`a.compareTo(b) > 0`
`a >= b`	`a.compareTo(b) >= 0`
`a == b`	`a.compareTo(b) == 0`
`a != b`	`a.compareTo(b) != 0`

Here is a compareTo() method for the Complex class in Example 9-2 that compares complex numbers by magnitude:

```
Complex.prototype.compareTo = function(that) {
    // If we aren't given an argument, or are passed a value that
    // does not have a magnitude() method, throw an exception
    // An alternative would be to return -1 or 1 in this case to say
    // that all Complex objects are always less than or greater than
    // any other values.
    if (!that || !that.magnitude || typeof that.magnitude != "function")
        throw new Error("bad argument to Complex.compareTo()");

    // This subtraction trick returns a value less than, equal to, or
    // greater than zero.  It is useful in many compareTo() methods.
    return this.magnitude() - that.magnitude();
}
```

One reason to compare instances of a class is so that arrays of those instances can be sorted into some order. The Array.sort() method accepts as an optional argument a comparison function that uses the same return-value conventions as the compareTo() method. Given the compareTo() method shown, it is easy to sort an array of Complex objects with code like this:

```
complexNumbers.sort(function(a,b) { return a.compareTo(b); });
```

Sorting is important enough that you should consider adding a static compare() method to any class for which you define a compareTo() instance method. One can easily be defined in terms of the other. For example:

```
Complex.compare = function(a,b) { return a.compareTo(b); };
```

With a method like this defined, sorting becomes simpler:

```
complexNumbers.sort(Complex.compare);
```

Notice that the `compareTo()` and `compare()` methods shown here were not included in the original Complex class of Example 9-2. That is because they are not *consistent with* the `equals()` method that was defined in that example. The `equals()` method says that two Complex objects are equal only if both their real and imaginary parts are the same. But the `compareTo()` method returns zero for any two complex numbers that have the same magnitude. Both the numbers 1+0*i* and 0+1*i* have the same magnitude, and these two values are equal according to `compareTo()` but not according to `equals()`. If you write `equals()` and `compareTo()` methods for the same Class, it is a good idea to make them consistent. Inconsistent notions of equality can be a pernicious source of bugs. Here is a `compareTo()` method that defines an ordering consistent with the existing `equals()` method:

```
// Compare complex numbers first by their real part.  If their real
// parts are equal, compare them by complex part
Complex.prototype.compareTo = function(that) {
    var result = this.x - that.x;   // compare real using subtraction
    if (result == 0)                // if they are equal...
        result = this.y - that.y;   //   then compare imaginary parts
    // Now our result is 0 if and only if this.equals(that)
    return result;
};
```

9.5 Superclasses and Subclasses

Java, C++, and other class-based object-oriented languages have an explicit concept of the *class hierarchy*. Every class can have a *superclass* from which it inherits properties and methods. Any class can be extended, or subclassed, so that the resulting *subclass* inherits its behavior. As shown previously, JavaScript supports prototype inheritance instead of class-based inheritance. Still, JavaScript analogies to the class hierarchy can be drawn. In JavaScript, the Object class is the most generic, and all other classes are specialized versions, or subclasses, of it. Another way to say this is that Object is the superclass of all the built-in classes, and all classes inherit a few basic methods from Object.

Recall that objects inherit properties from the prototype object of their constructor. How do they also inherit properties from the Object class? Remember that the prototype object is itself an object; it is created with the `Object()` constructor. This means the prototype object itself inherits properties from `Object.prototype`! Prototype-based inheritance is not limited to a single prototype object; instead, a chain of prototype objects is involved. Thus, a Complex object inherits properties from `Complex.prototype` and from `Object.prototype`. When you look up a property in a Complex object, the object itself is searched first. If the property is not found, the `Complex.prototype` object is searched next. Finally, if the property is not found in that object, the `Object.prototype` object is searched.

Note that because the Complex prototype object is searched before the Object prototype object, properties of `Complex.prototype` hide any properties with the same name in `Object.prototype`. For example, in the Complex class of Example 9-2, a `toString()` method was defined in the `Complex.prototype` object. `Object.prototype` also defines a method with this name, but Complex objects never see it because the definition of `toString()` in `Complex.prototype` is found first.

The classes shown so far in this chapter are all direct subclasses of Object. When necessary, however, it is possible to subclass any other class. Recall the `Rectangle` class shown earlier in the chapter, for example. It had properties that represent the width and height of the rectangle but no properties describing its position. Suppose you want to create a subclass of Rectangle in order to add fields and methods related to the position of the rectangle. To do this, use the `heir()` method of Section 9.2 to create a prototype for the new class that inherits from `Rectangle.prototype`.

Example 9-3 repeats the definition of a simple Rectangle class and then extends it to define a PositionedRectangle class.

Example 9-3. Subclassing a JavaScript class

```
// Here is a simple Rectangle class.
// It has a width and height and can compute its own area
function Rectangle(w, h) {
    this.width = w;
    this.height = h;
}
Rectangle.prototype.area = function() { return this.width * this.height; }

// Here is how we might subclass it
// First, we define a the subclass constructor.
function PositionedRectangle(w, h, x, y) {
    // First, invoke the superclass constructor on the new object
    // so that it can initialize the width and height.
    // We use the call method so that we invoke the constructor as a
    // method of the object to be initialized.
    // This is called constructor chaining.
    Rectangle.call(this, w, h);

    // Now store the position of the upper-left corner of the rectangle
    this.x = x;
    this.y = y;
}

// Create a prototype for the subclass that inherits from the prototype
// of the superclass. We do this with the heir() function.
function heir(p) {
    function f(){}
    f.prototype = p;
    return new f();
}
PositionedRectangle.prototype = heir(Rectangle.prototype);
```

Example 9-3. Subclassing a JavaScript class (continued)

```
// Since the subclass prototype object was created with the heir() function,
// it does not have a meaningful constructor property.  So set that now.
PositionedRectangle.prototype.constructor = PositionedRectangle;

// Now that we've configured the prototype object for our subclass,
// we can add instance methods to it.
PositionedRectangle.prototype.contains = function(x,y) {
    return (x > this.x &&
            x < this.x + this.width &&
            y > this.y &&
            y < this.y + this.height);
};
```

As you can see from Example 9-3, creating a subclass in JavaScript is not as simple as creating a class that inherits directly from Object. First, there is the issue of invoking the superclass constructor from the subclass constructor. Take care when you do this that the superclass constructor is invoked as a method of the newly created object. (You may want to review Section 8.6.4 on the `call()` and `apply()` methods of functions.) Next, there are the tricks required to set the prototype object of the subclass constructor. You must explicitly create this prototype object as an instance of the superclass, then explicitly set the constructor property of the prototype object. Optionally, you may also want to delete any properties that the superclass constructor created in the prototype object because what's important are the properties that the prototype object inherits from *its* prototype.

Having defined this PositionedRectangle class, you can use it with code like this:

```
var r = new PositionedRectangle(2,2,2,2);
print(r.contains(3,3));  // invoke an instance method
print(r.area());         // invoke an inherited instance method

// Use the instance fields of the class:
print(r.x + ", " + r.y + ", " + r.width + ", " + r.height);

// Our object is an instance of all 3 of these classes
print(r instanceof PositionedRectangle &&
      r instanceof Rectangle &&
      r instanceof Object);
```

9.5.1 Constructor Chaining

In the example just shown, we saw that the `PositionedRectangle()` constructor function needed to explicitly invoke the superclass constructor function. This is called *constructor chaining* and is quite common when creating subclasses. If you prefer not to explicitly refer to to the superclass constructor from the subclass constructor, you can add a property named `superclass` to the subclass constructor:

```
PositionedRectangle.superclass = Rectangle;
```

With this property defined, you can use `arguments.callee.superclass` in place of an explicit reference to `Rectangle`, and chain to the superclass constructor with boilerplate code like this:

```
arguments.callee.superclass.call(this, w, h)
```

`arguments.callee` always refers to the currently executing function. (See section 8.2.2.1). While you might be tempted to use `this.constructor.superclass` instead, it won't work: if someone creates a subclass of `PositionedRectangle`, then `this.constructor` will refer to the new subclass constructor, not to `PositionedRectangle`.

9.5.2 Invoking Overridden Methods

When a subclass defines a method that has the same name as a method in the superclass, the subclass *overrides* that method. This is a relatively common thing to do when creating subclasses of existing classes. Anytime you define a `toString()` method for a class, you override the `toString()` method of Object, for example.

A method that overrides another often wants to augment the functionality of the overridden method instead of replacing it altogether. To do that, a method must be able to invoke, or chain to, the method that it overrides

Let's consider an example. Suppose the `Rectangle` class had defined a `toString()` method (as it should have in the first place):

```
Rectangle.prototype.toString = function() {
    return "[" + this.width + "," + this.height + "]";
}
```

If you give Rectangle a `toString()` method, you really must override that method in PositionedRectangle so that instances of the subclass have a string representation that reflects all their properties, not just their width and height properties. PositionedRectangle is a simple enough class that its `toString()` method can just return the values of all properties. But for the sake of example, let's handle the position properties and delegate to its superclass for the `width` and `height` properties. Here is what the code might look like:

```
PositionedRectangle.prototype.toString = function() {
    return "(" + this.x + "," + this.y + ") " +       // our fields
           Rectangle.prototype.toString.call(this);   // chain to superclass
}
```

The superclass's implementation of `toString()` is a property of the superclass's prototype object. Note that you can't invoke it directly. Invoke it with `call()` so that you can specify the object on which it should be called. The `toString()` method we're using as an example does not take arguments, but if you want to pass arguments to an overridden method add the arguments to the invocation of `call()`.

Note that invoking the `toString()` method through `Rectangle.prototype` works even if the Rectangle class is modified to remove its `toString()` method. In that case the prototype object of the Rectangle class inherits to `toString()` method of Object, and the code will chain to that method instead.

We will return to chaining in Example 9-10, which simplifies method chaining using `arguments.callee` as we did for constructor chaining above.

9.6 Extending Without Inheriting

The discussion of creating subclasses earlier in this chapter explains how to create a new class that inherits the methods of another. JavaScript is such a flexible language that subclassing and inheritance are not the only way to extend a class. Since JavaScript functions are data values, you can simply copy (or "borrow") the functions from one class for use in another. Example 9-4 shows a function that borrows all the methods in one class and refers to them in the prototype object of another class.

Example 9-4. Borrowing methods from one class for use by another

```
// Borrow methods from one class for use by another.
// The arguments should be the constructor functions for the classes.
// Methods of built-in types such as Object, Array, Date, and RegExp are
// not enumerable and cannot be borrowed with this method.
function borrowMethods(borrowFrom, addTo) {
    var from = borrowFrom.prototype;  // prototype object to borrow from
    var to = addTo.prototype;         // prototype object to extend

    for(m in from) {  // Loop through all properties of the prototye
        if (typeof from[m] != "function") continue; // ignore nonfunctions
        to[m] = from[m];  // borrow the method
    }
}
```

Many methods are tied strongly to the class that defines them, and it makes no sense to try to use them in other classes. But it is possible to write some methods generically so that they are suitable for use by any class, or by any class that defines certain properties. Example 9-5 includes two classes that do nothing but define useful methods that other classes can borrow. Classes like these that are designed for borrowing are called *mixin classes* or *mixins*.

Example 9-5. Mixin classes with generic methods for borrowing

```
// This class isn't good for much on its own. But it does define a
// generic toString() method that may be of interest to other classes.
function GenericToString() {}
GenericToString.prototype.toString = function() {
    var props = [];
    for(var name in this) {
        if (!this.hasOwnProperty(name)) continue;
        var value = this[name];
        var s = name + ":"
        switch(typeof value) {
```

Example 9-5. Mixin classes with generic methods for borrowing (continued)

```
        case 'function':
            s += "function";
            break;
        case 'object':
            if (value instanceof Array) s += "array"
            else s += value.toString();
            break;
        default:
            s += String(value);
            break;
        }
        props.push(s);
    }
    return "{" + props.join(", ") + "}";
}

// This mixin class defines an equals() method that can compare
// simple objects for equality.
function GenericEquals() {}
GenericEquals.prototype.equals = function(that) {
    if (this == that) return true;

    // this and that are equal only if this has all the properties of
    // that and doesn't have any additional properties
    // Note that we don't do deep comparison.  Property values
    // must be === to each other.  So properties that refer to objects
    // must refer to the same object, not objects that are equals()
    var propsInThat = 0;
    for(var name in that) {
        propsInThat++;
        if (this[name] !== that[name]) return false;
    }

    // Now make sure that this object doesn't have additional props
    var propsInThis = 0;
    for(name in this) propsInThis++;

    // If this has additional properties, then they are not equal
    if (propsInThis != propsInThat) return false;

    // The two objects appear to be equal.
    return true;
}
```

Here is a simple Rectangle class that borrows the `toString()` and `equals()` methods defined by the mixin classes:

```
// Here is a simple Rectangle class.
function Rectangle(x, y, w, h) {
    this.x = x;
    this.y = y;
    this.width = w;
    this.height = h;
}
```

```
Rectangle.prototype.area = function() { return this.width * this.height; }

// Borrow some more methods for it
borrowMethods(GenericEquals, Rectangle);
borrowMethods(GenericToString, Rectangle);
```

Neither of the mixins shown have a constructor function, but it is possible to borrow constructors as well. In the following code, a new class is created named ColoredRectangle. It inherits rectangle functionality from Rectangle and borrows a constructor and a method from a mixin named Colored:

```
// This mixin has a method that depends on its constructor.  Both the
// constructor and the method must be borrowed.
function Colored(c) { this.color = c; }
Colored.prototype.getColor = function() { return this.color; }

// Define the constructor for a new class.
function ColoredRectangle(x, y, w, h, c) {
    Rectangle.call(this,x,y,w,h);  // Invoke superclass constructor
    Colored.call(this, c);         // and borrow the Colored constructor
}

// Set up the prototype object to inherit methods from Rectangle
ColoredRectangle.prototype = heir(Rectangle.prototype);
ColoredRectangle.prototype.constructor = ColoredRectangle;

// And borrow the methods of Colored for our new class
borrowMethods(Colored, ColoredRectangle);
```

The ColoredRectangle class extends (and inherits methods from) Rectangle and borrows methods from Colored. Rectangle itself inherits from Object and borrows from GenericEquals and GenericToString. Although any kind of strict analogy is impossible, you can think of this as a kind of multiple inheritance. Since the ColoredRectangle class borrows the methods of Colored, instances of ColoredRectangle can be considered instances of Colored as well. The instanceof operator will not report this, but in Section 9.7.3, we'll develop a more general method for determining whether an object inherits from *or* borrows from a specified class.

9.7 Determining Object Type

JavaScript is loosely typed, and JavaScript objects are even more loosely typed. There are a number of techniques you can use to determine the type of an arbitrary value in JavaScript.

The most obvious technique is the typeof operator, of course (see Section 5.10.2 for details). typeof is useful primarily for distinguishing primitive types from objects. There are a few quirks to typeof. First, remember that typeof null is "object", while typeof undefined is "undefined". Also, the type of any array is "object" because all arrays are objects. However, the type of any function is "function", even though functions are objects, too.

9.7.1 instanceof and constructor

Once you have determined that a value is an object rather than a primitive value or a function, you can use the `instanceof` operator to learn more about it. For example, if `x` is an array, the following evaluates to `true`:

```
x instanceof Array
```

The left side of `instanceof` is the value to be tested, and the right side should be a constructor function that defines a class. Note that an object is an instance of its own class and of any superclasses. So, for any object `o`, `o instanceof Object` is always `true`. Interestingly, `instanceof` works for functions, so for any function `f`, all these expressions are `true`:

```
typeof f == "function"
f instanceof Function
f instanceof Object
```

If you want to test whether an object is an instance of one specific class and not an instance of some subclass, you can check its `constructor` property. Consider the following code:

```
var d = new Date();                      // A Date object; Date extends Object
var isobject = d instanceof Object;      // evaluates to true
var realobject = d.constructor==Object; // evaluates to false
```

9.7.2 Object.toString( ) for Object Typing

One shortcoming of the `instanceof` operator and the `constructor` property is that they allow you to test an object only against classes you already know about. They aren't useful to inspect unknown objects, as you might do when debugging, for example. A useful trick that uses the default implementation of the `Object.toString()` method can help in this case.

As shown in Chapter 7, Object defines a default `toString()` method. Any class that does not define its own method inherits this default implementation. An interesting feature of the default `toString()` method is that it reveals some internal type information about built-in objects. The ECMAScript specification requires that this default `toString()` method always returns a string of the form:

```
[object class]
```

`class` is the internal type of the object and usually corresponds to the name of the constructor function for the object. For example, arrays have a *`class`* of "Array", functions have a *`class`* of "Function", and Date objects have a *`class`* of "Date". The built-in Math object has a *`class`* of "Math", and all Error objects (including instances of the various Error subclasses) have a *`class`* of "Error". Client-side JavaScript objects and any other objects defined by the JavaScript implementation have an implementation-defined *`class`* (such as "Window", "Document", or "Form"). Objects of user-defined types, such as the Circle and Complex classes defined earlier

in this chapter, always have a *class* of "Object", however, so this `toString()` technique is useful only for built-in object types.

Since most classes override the default `toString()` method, you can't invoke it directly on an object and expect to find its class name. Instead, you refer to the default function explicitly in `Object.prototype` and use `apply()` to invoke it on the object whose type you're interested in:

```
Object.prototype.toString.apply(o); // Always invokes the default toString()
```

This technique is used in Example 9-6 to define a function that provides enhanced "type of" functionality. As noted earlier, the `toString()` method does not work for user-defined classes, so in this case, the function checks for a string-valued property of the constructor named `classname` and returns its value if it exists.

Example 9-6. Enhanced typeof testing

```
function getType(x) {
    // If x is null, return "null"
    if (x == null) return "null";

    // Next try the typeof operator
    var t = typeof x;
    // If the result is not vague, return it
    if (t != "object")  return t;

    // Otherwise, x is an object. Use the default toString() method to
    // get the class value of the object.
    var c = Object.prototype.toString.apply(x);  // Returns "[object class]"
    c = c.substring(8, c.length-1);             // Strip off "[object" and "]"

    // If the class is not a vague one, return it.
    if (c != "Object") return c;

    // If we get here, c is "Object".  Check to see if
    // the value x is really just a generic object.
    if (x.constructor == Object) return c;  // Okay the type really is "Object"

    // For user-defined classes, look for a string-valued property of
    // the constructor named classname.
    if (x.constructor && x.constructor.classname && // If class has a name
        typeof x.constructor.classname == "string") //   and it is a string
        return x.constructor.classname;            //   then return it.

    // If we really can't figure it out, say so.
    return "<unknown type>";
}
```

9.7.3 Duck Typing

There is an old saying: "If it walks like a duck and quacks like a duck, it's a duck!" Translated into JavaScript, this aphorism is not nearly so evocative. Try it this way: "If it implements all the methods defined by a class, it is an instance of that class." In flexible, loosely typed languages like JavaScript, this is called *duck typing*: if an object has the properties defined by class X, you can treat it as an instance of class X, even if it was not actually created with the X( ) constructor function.*

Duck typing is particularly useful in conjunction with classes that "borrow" methods from other classes. Earlier in the chapter, a Rectangle class borrowed the implementation of an equals( ) method from another class named GenericEquals. Thus, you can consider any Rectangle instance to also be an instance of GenericEquals. The instanceof operator will not report this, but you can define a method that will. Example 9-7 shows how.

Example 9-7. Testing whether an object borrows the methods of a class

```
// Return true if each of the method properties in c.prototype have been
// borrowed by o. If o is a function rather than an object, we
// test the prototype of o rather than o itself.
// Note that this function requires methods to be copied, not
// reimplemented.  If a class borrows a method and then overrides it,
// this method will return false.
function borrows(o, c) {
    // If we are an instance of something, then of course we have its methods
    if (o instanceof c) return true;

    // It is impossible to test whether the methods of a built-in type have
    // been borrowed, since the methods of built-in types are not enumerable.
    // We return undefined in this case as a kind of "I don't know" answer
    // instead of throwing an exception. Undefined behaves much like false,
    // but can be distinguished from false if the caller needs to.
    if (c == Array || c == Boolean || c == Date || c == Error ||
        c == Function || c == Number || c == RegExp || c == String)
        return undefined;

    if (typeof o == "function") o = o.prototype;
    var proto = c.prototype;
    for(var p in proto) {
        // Ignore properties that are not functions
        if (typeof proto[p] != "function") continue;
        if (o[p] != proto[p]) return false;
    }
    return true;
}
```

* The term "duck typing" has been popularized by the Ruby programming language. A more formal name is *allomorphism*.

The borrows( ) method of Example 9-7 is relatively strict: it requires the object o to have exact copies of the methods defined by the class c. True duck typing is more flexible: o should be considered an instance of c as long as it provides methods that look like methods of c. In JavaScript, "look like" means "have the same name as" and (perhaps) "are declared with the same number of arguments as." Example 9-8 shows a method that tests for this.

Example 9-8. Testing whether an object provides methods

```
// Return true if o has methods with the same name and arity as all
// methods in c.prototype. Otherwise, return false.  Throws an exception
// if c is a built-in type with nonenumerable methods.
function provides(o, c) {
    // If o actually is an instance of c, it obviously looks like c
    if (o instanceof c) return true;

    // If a constructor was passed instead of an object, use its prototype
    if (typeof o == "function") o = o.prototype;

    // The methods of built-in types are not enumerable, and we return
    // undefined.  Otherwise, any object would appear to provide any of
    // the built-in types.
    if (c == Array || c == Boolean || c == Date || c == Error ||
        c == Function || c == Number || c == RegExp || c == String)
        return undefined;

    var proto = c.prototype;
    for(var p in proto) {  // Loop through all properties in c.prototype
        // Ignore properties that are not functions
        if (typeof proto[p] != "function") continue;
        // If o does not have a property by the same name, return false
        if (!(p in o)) return false;
        // If that property is not a function, return false
        if (typeof o[p] != "function") return false;
        // If the two functions are not declared with the same number
        // of arguments, return false.
        if (o[p].length != proto[p].length) return false;
    }
    // If all the methods check out, we can finally return true.
    return true;
}
```

As an example of when duck typing and the provides( ) method are useful, consider the compareTo( ) method described in Section 9.4.3. compareTo( ) is not a method that lends itself to borrowing, but it would still be nice if we could easily test for objects that are comparable with the compareTo( ) method. To do this, define a Comparable class:

```
function Comparable( ) {}
Comparable.prototype.compareTo = function(that) {
    throw "Comparable.compareTo( ) is abstract.  Don't invoke it!";
}
```

This Comparable class is *abstract*: its method isn't designed to actually be invoked but simply to define an API. With this class defined, however, you can check if two objects can be compared like this:

```
// Check whether objects o and p can be compared
// They must be of the same type, and that type must be comparable
if (o.constructor == p.constructor && provides(o, Comparable)) {
    var order = o.compareTo(p);
}
```

Note that both the `borrows()` and `provides()` functions presented in this section return `undefined` if passed any core JavaScript built-in type, such as Array. This is because the properties of the prototype objects of the built-in types are not enumerable with a `for/in` loop. If those functions did not explicitly check for built-in types and return `undefined`, they would think that these built-in types have no methods and would always return `true` for built-in types.

The Array type is one that is worth considering specially, however. Recall from Section 7.8 that many array algorithms (such as iterating over the elements) can work fine on objects that are not true arrays but are array-like. Another application of duck typing is to determine which objects look like arrays. Example 9-9 shows one way to do it.

Example 9-9. Testing for array-like objects

```
function isArrayLike(x) {
    if (x instanceof Array) return true; // Real arrays are array-like
    if (!("length" in x)) return false;  // Arrays must have a length property
    if (typeof x.length != "number") return false;  // Length must be a number
    if (x.length < 0) return false;                 // and nonnegative
    if (x.length > 0) {
        // If the array is nonempty, it must at a minimum
        // have a property defined whose name is the number length-1
        if (!((x.length-1) in x)) return false;
    }
    return true;
}
```

9.8 Example: A defineClass( ) Utility Method

This chapter ends with a `defineClass()` utility function that ties together the previous discussions of constructors, prototypes, subclassing, and borrowing and chaining. Example 9-10 defines the function and Example 9-11 shows how it can be used.

Example 9-10. A utility function for defining classes

```
/**
 * defineClass() -- a utility function for defining JavaScript classes.
 *
 * This function expects a single object as its only argument. It defines
 * a new JavaScript class based on the data in that object and returns the
 * constructor function of the new class.
```

Example 9-10. 'A utility function for defining classes

```
/**
 *
 * The object passed as an argument should have some or all of the
 * following properties:
 *
 *      name: the name of the class being defined.
 *            If specified, this value will be stored in the classname
 *            property of the returned constructor object.
 *
 *    extend: The constructor of the class to be extended. The returned
 *            constructor automatically chains to this function. This value
 *            is stored in the superclass property of the constructor object.
 *
 *      init: The initialization function for the class. If defined, the
 *            constructor will pass all of its arguments to this function.
 *            The constructor also automatically invokes the superclass
 *            constructor with the same arguments, so this function must expect
 *            the same arguments, in the same order, as the superclass
 *            constructor, and can add additional arguments at the end.
 *
 *   methods: An object that specifies the instance methods (and other
 *            non-method properties for the class.  The properties of
 *            this object become properties of the prototype.  Methods
 *            are given an overrides property for chaining.  They can
 *            call "chain(this, arguments)" to invoke the method they
 *            override. This function adds properties to the methods in
 *            this object, so you may not pass the same method in two
 *            invocations of defineClass().
 *
 *   statics: An object that specifies the static methods (and other static
 *            properties) for the class.  The properties of this object become
 *            properties of the constructor function.
 **/
function defineClass(data) {
    // Extract some properties from the argument object
    var extend = data.extend;
    var superclass = extend || Object;
    var init = data.init;
    var classname = data.name || "Unnamed class";
    var methods = data.methods || {};
    var statics = data.statics || {};

    // Make a constructor function that chains to the superclass constructor
    // and then calls the initialization method of this class.
    // This will become the return value of this defineClass() method.
    var constructor = function() {
        if (extend) extend.apply(this, arguments); // Initialize superclass
        if (init) init.apply(this, arguments);     // Initialize ourself
    };

    // Copy static properties to the constructor function
    if (data.statics)
        for(var p in data.statics) constructor[p] = data.statics[p];
```

Example 9-10. `A utility function for defining classes

```
/**

    // Set superclass and classname properties of the constructor
    constructor.superclass = superclass;
    constructor.classname = classname;

    // Create the object that will be the prototype for the class.
    // This new object must inherit from the superclass prototype.
    var proto = (superclass == Object) ? {} : heir(superclass.prototype);

    // Copy instance methods (and other properties) to the prototype object.
    for(var p in methods) {            // For each name in methods object
        if (p == "toString") continue; // Handled below
        var m = methods[p];            // This is the value to copy
        if (typeof m == "function") {  // If it is a function
            m.overrides = proto[p];    // Remember anything it overrides
            m.name = p;                // Tell it what its name is
            m.owner = constructor;     // Tell it what class owns it.
        }
        proto[p] = m;                  // Then store in the prototype
    }

    // In IE, a for/in loop won't enumerate properties that have the same name
    // as non-enumerable Object methods like toString(). As a partial
    // work-around, we handle the toString method specially
    if (methods.hasOwnProperty("toString")) { // IE DontEnum bug
        methods.toString.overrides = proto.toString;
        methods.toString.name = "toString";
        methods.toString.owner = constructor;
        proto.toString = methods.toString;
    }

    // All objects should know who their constructor was
    proto.constructor = constructor;

    // And the constructor must know what its prototype is
    constructor.prototype = proto;

    // Finally, return the constructor function
    return constructor;
}

/**
 * Return a new object with p as its prototype
 */
function heir(p) {
    function h(){}
    h.prototype=p;
    return new h();
}

/**
 * Chain from the calling function to the function on its overrides property.
 * Invoke that method on the first argument. The second argument must be the
```

Example 9-10. 'A utility function for defining classes

```
/**
 * arguments object of the calling function; its callee property is used to
 * determine what function is doing the chaining. The third argument is an
 * optional array of values to pass to the overridden method.  If omitted,
 * the second argument is used instead, passing all of the caller's arguments
 * on to the overridden method.
 *
 * This method returns the return value of the overridden method or
 * throws "ChainError" if no overridden method could be found
 *
 * Typical invocation:     chain(this, arguments)
 * To pass different args: chain(this, arguments, [w, h])
 */
function chain(o, args, pass) {
    var f = args.callee;          // The calling function.
    var g = f.overrides;          // The function it chains to.
    var a = pass || args;         // The arguments we'll pass to s
    if (g) return g.apply(o, a); // Call o.g(a) and return its value as ours.
    else throw "ChainError"       // Complain if nothing to override
}
```

Example 9-11 shows sample code that uses the defineClass( ) function.

Example 9-11. Using the defineClass() function .

```
// A very simple Rectangle class
var Rectangle = defineClass({
    name: "Rectangle",
    init: function(w,h) {
        this.w = w;
        this.h = h;
    },
    methods: {
        area: function() { return this.w * this.h; },
        toString: function() { return "[" + this.w + "," + this.h + "]" }
    }
});

// A subclass of Rectangle
var PositionedRectangle = defineClass({
    name: "PositionedRectangle",
    extend: Rectangle,
    init: function(w,h,x,y) {
        // Automatic chain here: Rectangle.call(this,w,h,x,y)
        this.x = x;
        this.y = y;
    },
    methods: {
        isInside: function(x,y) {
            return x > this.x && x < this.x + this.w &&
                y > this.y && y < this.y + this.h;
        },
        toString: function() {
```

Example 9-11. Using the defineClass() function (continued).

```
// A very simple Rectangle class
            return chain(this, arguments) + "(" + this.x + "," + this.y + ")";
        }
    }
});

var ColoredRectangle = defineClass({
    name: "ColoredRectangle",
    extend: PositionedRectangle,
    init: function(w,h,x,y,c) { this.c = c; },
    methods: {
        toString: function() { return this.c + ": " + chain(this,arguments)}
    }
});
```

CHAPTER 13

JavaScript in Web Browsers

The first part of this book described the core JavaScript language. Part II moves on to JavaScript as used within web browsers, commonly called client-side JavaScript.* Most of the examples you've seen so far, while legal JavaScript code, have no particular context; they are JavaScript fragments that run in no specified environment. This chapter provides that context. It starts with an overview of the web browser programming environment. Next, it discusses how to actually embed JavaScript code within HTML documents, using `<script>` tags, HTML event handler attributes, and JavaScript URLs. These sections on embedding JavaScript are followed by a section that explains the client-side JavaScript execution model: how and when web browsers run JavaScript code. Next are sections that cover three important topics in JavaScript programming: compatibility, accessibility, and security. The chapter concludes with an short description of web-related embeddings of the JavaScript language other than client-side JavaScript.

When JavaScript is embedded in a web browser, the browser exposes a powerful and diverse set of capabilities and allows them to be scripted. The chapters that follow Chapter 13 each focus on one major area of client-side JavaScript functionality:

- Chapter 14, *Scripting Browser Windows*, explains how JavaScript can script web browser windows by, for example, opening and closing windows, displaying dialog boxes, causing windows to load specified URLs, or causing windows to go back or forward in their browsing history. This chapter also covers other, miscellaneous features of client-side JavaScript that happen to be associated with the Window object in client-side JavaScript.
- Chapter 15, *Scripting Documents*, explains how JavaScript can interact with the document content displayed within a web browser window and how it can find, insert, delete, or alter content within a document.

* The term *client-side JavaScript* is left over from the days when JavaScript was used in only two places: web browsers (clients) and web servers. As JavaScript is adopted as a scripting language in more and more environments, the term *client-side* makes less and less sense because it doesn't specify the client side of *what*. Nevertheless, I'll continue to use the term in this book.

- Chapter 16, *Cascading Style Sheets and Dynamic HTML*, covers the interaction of JavaScript and CSS and shows how JavaScript code can alter the presentation of a document by scripting CSS styles, classes, and stylesheets. One particularly potent result of combining scripting with CSS is Dynamic HTML (or DHTML) in which HTML content can be hidden and shown, moved, and even animated.
- Chapter 17, *Events and Event Handling*, explains events and event handling and shows how JavaScript adds interactivity to a web page by allowing it to respond to a user's input.
- Chapter 18, *Forms and Form Elements*, covers forms within HTML documents and shows how JavaScript can gather, validate, process, and submit user input with forms.
- Chapter 19, *Cookies and Client-Side Persistence*, shows how JavaScript scripts can persistently store data using HTTP cookies.
- Chapter 20, *Scripting HTTP*, introduces HTTP scripting (commonly known as Ajax) and demonstrates how JavaScript can communicate with web servers.
- Chapter 21, *JavaScript and XML*, shows how JavaScript can create, load, parse, transform, query, serialize, and extract information from XML documents.
- Chapter 22, *Scripted Client-Side Graphics*, demonstrates common JavaScript image-manipulation techniques that can create image rollovers and animations in web pages. It also demonstrates several techniques for dynamically drawing vector graphics under JavaScript control.
- Chapter 23, *Scripting Java Applets and Flash Movies*, explains how JavaScript can interact with Java applets and Flash movies embedded in a web page.

13.1 The Web Browser Environment

To understand client-side JavaScript, you must understand the programming environment provided by a web browser. The following sections introduce three important features of that programming environment:

- The Window object that serves as the global object and global execution context for client-side JavaScript code
- The client-side object hierarchy and the Document Object Model that forms a part of it
- The event-driven programming model

These sections are followed by a discussion of the proper role of JavaScript in web application development.

13.1.1 The Window as Global Execution Context

The primary task of a web browser is to display HTML documents in a window. In client-side JavaScript, the Document object represents an HTML document, and the Window object represents the browser window (or frame) that displays the document. While the Document and Window objects are both important to client-side JavaScript, the Window object is more important for one substantial reason: the Window object is the global object in client-side programming.

Recall from Chapter 4 that in every implementation of JavaScript there is always a global object at the head of the scope chain; the properties of this global object are global variables. In client-side JavaScript, the Window object is the global object. The Window object defines a number of properties and methods that allow you to manipulate the web browser window. It also defines properties that refer to other important objects, such as the `document` property for the Document object. Finally, the Window object has two self-referential properties, `window` and `self`. You can use either global variable to refer directly to the Window object.

Since the Window object is the global object in client-side JavaScript, all global variables are defined as properties of the window. For example, the following two lines of code perform essentially the same function:

```
var answer = 42;     // Declare and initialize a global variable
window.answer = 42;  // Create a new property of the Window object
```

The Window object represents a web browser window (or a frame within a window; in client-side JavaScript, top-level windows and frames are essentially equivalent). It is possible to write applications that use multiple windows (or frames). Each window involved in an application has a unique Window object and defines a unique execution context for client-side JavaScript code. In other words, a global variable declared by JavaScript code in one window is not a global variable within a second window. However, JavaScript code in the second window *can* access a global variable of the first window, subject to certain security restrictions. These issues are considered in detail in Chapter 14.

13.1.2 The Client-Side Object Hierarchy and the DOM

The Window object is the key object in client-side JavaScript. All other client-side objects are accessed via this object. For example, every Window object defines a `document` property that refers to the Document object associated with the window and a `location` property that refers to the Location object associated with the window. When a web browser displays a framed document, the `frames[]` array of the top-level Window object contains references to the Window objects that represent the frames. Thus, in client-side JavaScript, the expression `document` refers to the Document object of the current window; the expression `frames[1].document` refers to the Document object of the second frame of the current window.

The Document object (and other client-side JavaScript objects) also have properties that refer to other objects. For example, every Document object has a `forms[]` array containing Form objects that represent any HTML forms appearing in the document. To refer to one of these forms, you might write:

```
window.document.forms[0]
```

To continue with the same example, each Form object has an `elements[]` array containing objects that represent the various HTML form elements (input fields, buttons, etc.) that appear within the form. In extreme cases, you can write code that refers to an object at the end of a whole chain of objects, ending up with expressions as complex as this one:

```
parent.frames[0].document.forms[0].elements[3].options[2].text
```

As shown earlier, the Window object is the global object at the head of the scope chain, and all client-side objects in JavaScript are accessible as properties of other objects. This means that there is a hierarchy of JavaScript objects, with the Window object at its root. Figure 13-1 shows this hierarchy.

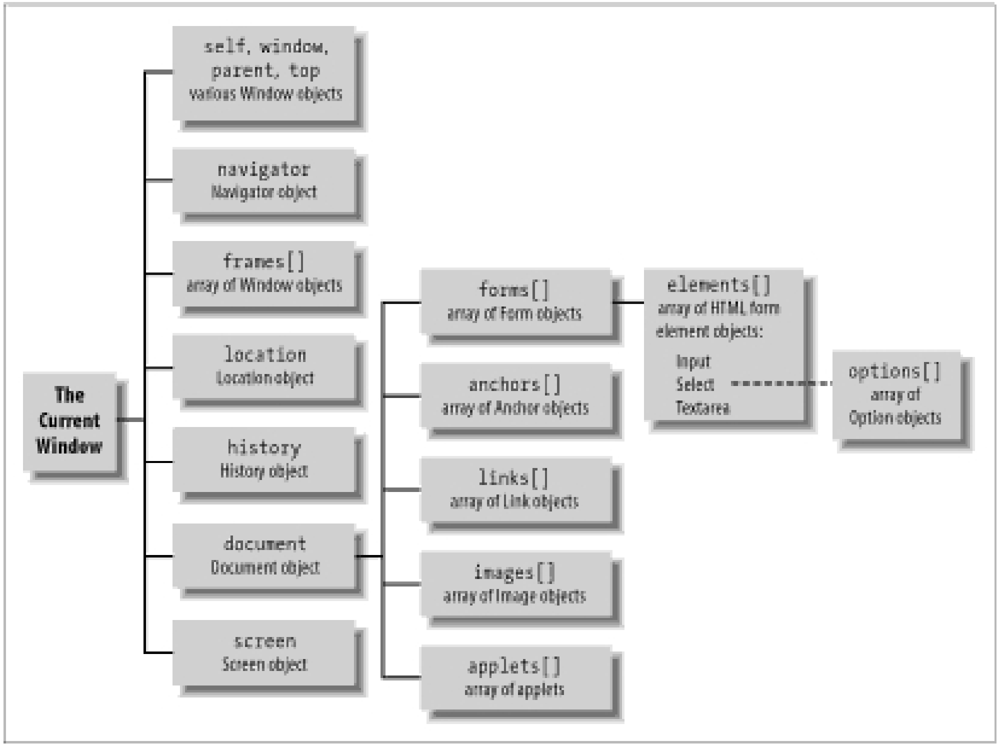


Figure 13-1. The client-side object hierarchy and Level 0 DOM

Note that Figure 13-1 shows just the object properties that refer to other objects. Most of the objects shown in the diagram have methods and properties other than those shown.

Many of the objects pictured in Figure 13-1 descend from the Document object. This subtree of the larger client-side object hierarchy is known as the document object model (DOM), which is interesting because it has been the focus of a standardization effort. Figure 13-1 illustrates the Document objects that have become de facto standards because they are consistently implemented by all major browsers. Collectively, they are known as the Level 0 DOM because they form a base level of document functionality that JavaScript programmers can rely on in all browsers. These basic Document objects are covered in Chapter 15, which also explains a more advanced document object model that has been standardized by the W3C. HTML forms are part of the DOM but are specialized enough that they are covered in their own chapter, Chapter 18.

13.1.3 The Event-Driven Programming Model

In the early days of computing, computer programs often ran in batch mode; they read in a batch of data, did some computation on that data, and then wrote out the results. Later, with time-sharing and text-based terminals, limited kinds of interactivity became possible; the program could ask the user for input, and the user could type in data. The computer then processed the data and displayed the results onscreen.

Nowadays, with graphical displays and pointing devices such as mice, the situation is different. Programs are generally event-driven; they respond to asynchronous user input in the form of mouse clicks and keystrokes in a way that depends on the position of the mouse pointer. A web browser is just such a graphical environment. An HTML document contains an embedded graphical user interface (GUI), so client-side JavaScript uses the event-driven programming model.

It is perfectly possible to write a static JavaScript program that does not accept user input and does exactly the same thing every time it is run. Sometimes this sort of program is useful. More often, however, you'll want to write dynamic programs that interact with the user. To do this, you must be able to respond to user input.

In client-side JavaScript, the web browser notifies programs of user input by generating *events*. There are various types of events, such as keystroke events, mouse motion events, and so on. When an event occurs, the web browser attempts to invoke an appropriate *event handler* function to respond to the event. Thus, to write dynamic, interactive client-side JavaScript programs, you must define appropriate event handlers and register them with the system, so that the browser can invoke them at appropriate times.

If you are not already accustomed to the event-driven programming model, it can take a little getting used to. In the old model, you wrote a single, monolithic block of code that followed some well-defined flow of control and ran to completion from beginning to end. Event-driven programming stands this model on its head. In event-driven programming, you write a number of independent (but mutually interacting) event handlers. You do not invoke these handlers directly but allow the system to invoke them at the appropriate times. Since they are triggered by the user's input, the handlers are invoked at unpredictable, asynchronous times. Much of the time, your program is not running at all but merely sitting, waiting for the system to invoke one of its event handlers.

The sections that follow explain how JavaScript code is embedded within HTML files. It shows how to define both static blocks of code that run synchronously from start to finish and event handlers that are invoked asynchronously by the system. Events and event handling are discussed again in Chapter 15, and then events are covered in detail in Chapter 17.

13.1.4 The Role of JavaScript on the Web

The introduction to this chapter included a list of the web browser capabilities that can be scripted with client-side JavaScript. Note, however, that listing what Java-Script *can* be used for is not the same as explaining what JavaScript *ought* to be used for. This section attempts to explain the proper role of JavaScript in web application development.

Web browsers display HTML-structured text styled with CSS stylesheets. HTML defines the content, and CSS supplies the presentation. Properly used, JavaScript adds *behavior* to the content and its presentation. The role of JavaScript is to enhance a user's browsing experience, making it easier to obtain or transmit information. The user's experience should not be dependent on JavaScript, but Java-Script can serve to facilitate that experience. JavaScript can do this in any number of ways. Here are some examples:

- Creating visual effects such as image rollovers that subtly guide a user and help with page navigation
- Sorting the columns of a table to make it easier for a user to find what he needs
- Hiding certain content and revealing details selectively as the user "drills down" into that content
- Streamlining the browsing experience by communicating directly with a web server so that new information can be displayed without requiring a complete page reload

13.1.5 Unobtrusive JavaScript

A new client-side programming paradigm known as *unobtrusive JavaScript* has been gaining currency within the web development community. As its name implies, this paradigm holds that JavaScript should not draw attention to itself; it should not obtrude.[*] It should not obtrude on users viewing a web page, on content authors creating HTML markup, or on web designers creating HTML templates or CSS stylesheets.

There is no specific formula for writing unobtrusive JavaScript code. However, a number of helpful practices, discussed elsewhere in this book, will put you on the right track.

The first goal of unobtrusive JavaScript is to keep JavaScript code separate from HTML markup. This keeps content separate from behavior in the same way that putting CSS in external stylesheets keeps content separate from presentation. To achieve this goal, you put all your JavaScript code in external files and include those files into your HTML pages with `<script src=>` tags (see Section 13.2.2 for details). If you are strict about the separation of content and behavior, you won't even include JavaScript code in the event-handler attributes of your HTML files. Instead, you will write JavaScript code (in an external file) that registers event handlers on the HTML elements that need them (Chapter 17 describes how to do this).

As a corollary to this goal, you should strive to make your external files of JavaScript code as modular as possible using techniques described in Chapter 10. This allows you to include multiple independent modules of code into the same web page without worrying about the variables and functions of one module overwriting the variables and functions of another.

The second goal of unobtrusive JavaScript is that it must degrade gracefully. Your scripts should be conceived and designed as enhancements to HTML content, but that content should still be available without your JavaScript code (as will happen, for example, when a user disables JavaScript in her browser). An important technique for graceful degradation is called *feature testing*: before taking any actions, your JavaScript modules should first ensure that the client-side features they require are available in the browser in which the code is running. Feature testing is a compatibility technique described in more detail in Section 13.6.3.

A third goal of unobtrusive JavaScript is that it must not degrade the accessibility of an HTML page (and ideally it should enhance accessibility). If the inclusion of JavaScript code reduces the accessibility of web pages, that JavaScript code has obtruded on the disabled users who rely on accessible web pages. JavaScript accessibility is described in more detail in Section 13.7.

[*] "Obtrude" is an obscure synonym for "intrude." The American Heritage dictionary cites: "To impose…on others with undue insistence or without invitation."

Other formulations of unobtrusive JavaScript may include other goals in addition to the three described here. One primary source from which to learn more about unobtrusive scripting is "The JavaScript Manifesto," published by the DOM Scripting Task Force at *http://domscripting.webstandards.org/?page_id=2*.

13.2 Embedding Scripts in HTML

Client-side JavaScript code is embedded within HTML documents in a number of ways:

- Between a pair of `<script>` and `</script>` tags
- From an external file specified by the `src` attribute of a `<script>` tag
- In an event handler, specified as the value of an HTML attribute such as `onclick` or `onmouseover`
- In a URL that uses the special `javascript:` protocol

This section covers `<script>` tags. Event handlers and JavaScript URLs are covered later in the chapter.

13.2.1 The <script> Tag

Client-side JavaScript scripts are part of an HTML file and are coded within `<script>` and `</script>` tags:

```
<script>
// Your JavaScript code goes here
</script>
```

In XHTML, the content of a `<script>` tag is treated like any other content. If your JavaScript code contains the `<` or `&` characters, these characters are interpreted as XML markup. For this reason, it is best to put all JavaScript code within a CDATA section if you are using XHTML:

```
<script><![CDATA[// Your JavaScript code goes here
]]></script>
```

A single HTML document may contain any number of `<script>` elements. Multiple, separate scripts are executed in the order in which they appear within the document (see the `defer` attribute in Section 13.2.4 for an exception, however). While separate scripts within a single file are executed at different times during the loading and parsing of the HTML file, they constitute part of the same JavaScript program: functions and variables defined in one script are available to all scripts that follow in the same file. For example, you can have the following script in the <head> of an HTML page:

```
<script>function square(x) { return x*x; }</script>
```

Later in the same HTML page, you can refer to the square( ) function, even though it's in a different script block. The context that matters is the HTML page, not the script block:*

```
<script>alert(square(2));</script>
```

Example 13-1 shows a sample HTML file that includes a simple JavaScript program. Note the difference between this example and many of the code fragments shown earlier in this book: this one is integrated with an HTML file and has a clear context in which it runs. Note also the use of a language attribute in the <script> tag; this is explained in Section 13.2.3.

Example 13-1. A simple JavaScript program in an HTML file

```
<html>
<head>
<title>Today's Date</title>
<script language="JavaScript">
// Define a function for later use
function print_todays_date() {
    var d = new Date();                   // Get today's date and time
    document.write(d.toLocaleString());   // Insert it into the document
}
</script>
</head>
<body>
The date and time are:<br>
<script language="JavaScript">
  // Now call the function we defined above
  print_todays_date();
</script>
</body>
</html>
```

Example 13-1 also demonstrates the document.write( ) function. Client-side JavaScript code can use this function to output HTML text into the document at the location of the script (see Chapter 15 for further details on this method). Note that the possibility that scripts can generate output for insertion into the HTML document means that the HTML parser must interpret JavaScript scripts as part of the parsing process. It is not possible to simply concatenate all script text in a document and run it as one large script after the document has been parsed because any script within a document may alter the document (see the discussion of the defer attribute in Section 13.2.4).

* The alert( ) function used here is a simple way to display output in client-side JavaScript: it converts its argument to a string and displays that string in a pop-up dialog box. See Section 14.5 for details on the alert( ) method, and see Example 15-9 for an alternative to alert( ) that does not pop up a dialog box that must be clicked away.

13.2.2 Scripts in External Files

The `<script>` tag supports a `src` attribute that specifies the URL of a file containing JavaScript code. It is used like this:

```
<script src="../../scripts/util.js"></script>
```

A JavaScript file typically has a *.js* extension and contains pure JavaScript, without `<script>` tags or any other HTML.

A `<script>` tag with the `src` attribute specified behaves exactly as if the contents of the specified JavaScript file appeared directly between the `<script>` and `</script>` tags. Any code or markup that appears between these tags is ignored. Note that the closing `</script>` tag is required even when the `src` attribute is specified, and there is no JavaScript between the `<script>` and `</script>` tags.

There are a number of advantages to using the `src` attribute:

- It simplifies your HTML files by allowing you to remove large blocks of JavaScript code from them—that is, it helps keep content and behavior separate. Using the `src` attribute is the cornerstone of unobtrusive JavaScript programming. (See Section 13.1.5 for more on this programming philosophy.)
- When you have a function or other JavaScript code used by several different HTML files, you can keep it in a single file and read it into each HTML file that needs it. This makes code maintenance much easier.
- When JavaScript functions are used by more than one page, placing them in a separate JavaScript file allows them to be cached by the browser, making them load more quickly. When JavaScript code is shared by multiple pages, the time savings of caching more than outweigh the small delay required for the browser to open a separate network connection to download the JavaScript file the first time it is requested.
- Because the `src` attribute takes an arbitrary URL as its value, a JavaScript program or web page from one web server can employ code exported by other web servers. Much Internet advertising relies on this fact.

This last point has important security implications. The same-origin security policy described in Section 13.8.2 prevents JavaScript in a document from one domain from interacting with content from another domain. However, notice that the origin of the script itself does not matter: only the origin of the document in which the script is embedded. Therefore, the same-origin policy does not apply in this case: JavaScript code can interact with the document in which it is embedded, even when the code has a different origin than the document. When you use the `src` attribute to include a script in your page, you are giving the author of that script (and the webmaster of the domain from which the script is loaded) complete control over your web page.

13.2.3 Specifying the Scripting Language

Although JavaScript was the original scripting language for the Web and remains the most common by far, it is not the only one. The HTML specification is language-agnostic, and browser vendors can support whatever scripting languages they choose. In practice, the only alternative to JavaScript is Microsoft's Visual Basic Scripting Edition,* which is supported by Internet Explorer.

Since there is more than one possible scripting language, you must tell the web browser what language your scripts are written in. This enables it to interpret the scripts correctly and to skip scripts written in languages that it does not know how to interpret. You can specify the default scripting language for a file with the HTTP `Content-Script-Type` header, and you can simulate this header with the HTML `<meta>` tag. To specify that all your scripts are in JavaScript (unless specified otherwise), just put the following tag in the `<head>` of all your HTML documents:

```
<meta http-equiv="Content-Script-Type" content="text/javascript">
```

In practice, browsers assume that JavaScript is the default scripting language even if your server omits the `Content-Script-Type` header and your pages omit the `<meta>` tag. If you do not specify a default scripting language, however, or wish to override your default, you should use the `type` attribute of the `<script>` tag:

```
<script type="text/javascript"></script>
```

The traditional MIME type for JavaScript programs is "text/javascript". Another type that has been used is "application/x-javascript" (the "x-" prefix indicates that it is an experimental, nonstandard type). RFC 4329 standardizes the "text/javascript" type because it is in common use. However, because JavaScript programs are not really text documents, it marks this type as obsolete and recommends "application/javascript" (without the "x-") instead. At the time of this writing, "application/javascript" is not well supported, however. Once it has become well supported, the most appropriate `<script>` and `<meta>` tags will be:

```
<script type="application/javascript"></script>
<meta http-equiv="Content-Script-Type" content="application/javascript">
```

When the `<script>` tag was first introduced, it was a nonstandard extension to HTML and did not support the `type` attribute. Instead, the scripting language was defined with the `language` attribute. This attribute simply specifies the common name of the scripting language. If you are writing JavaScript code, use the `language` attribute as follows:

```
<script language="JavaScript">
    // JavaScript code goes here
</script>
```

* Also known as VBScript. The only browser that supports VBScript is Internet Explorer, so scripts written in this language are not portable. VBScript interfaces with HTML objects the same way JavaScript does, but the core language itself has a different syntax than JavaScript. VBScript is not documented in this book.

And if you are writing a script in VBScript, use the attribute like this:

```
<script language="VBScript">
    ' VBScript code goes here (' is a comment character like // in JavaScript)
</script>
```

The HTML 4 specification standardized the `<script>` tag, but it deprecated the `language` attribute because there is no standard set of names for scripting languages. Sometimes you'll see `<script>` tags that use the `type` attribute for standards compliance and the `language` attribute for backward compatibility with older browsers:

```
<script type="text/javascript" language="JavaScript"></script>
```

The `language` attribute is sometimes used to specify the version of JavaScript in which a script is written, with tags like these:

```
<script language="JavaScript1.2"></script>
<script language="JavaScript1.5"></script>
```

In theory, web browsers ignore scripts written in versions of JavaScript that they do not support. That is, an old browser that does not support JavaScript 1.5 will not attempt to run a script that has a `language` attribute of "JavaScript1.5". Older web browsers respect this version number, but because the core JavaScript language has remained stable for a number of years, many newer browsers ignore any version number specified with the `language` attribute.

13.2.4 The defer Attribute

As mentioned earlier, a script may call the `document.write( )` method to dynamically add content to a document. Because of this, when the HTML parser encounters a script, it must normally stop parsing the document and wait for the script to execute. The HTML 4 standard defines a `defer` attribute of the `<script>` tag to address this problem.

If you write a script that does not produce any document output—for example, a script that defines a function but never calls `document.write( )`—you may use the `defer` attribute in the `<script>` tag as a hint to the browser that it is safe to continue parsing the HTML document and defer execution of the script until it encounters a script that cannot be deferred. Deferring a script is particularly useful when it is loaded from an external file; if it is not deferred, the browser must wait until the script has loaded before it can resume parsing the containing document. Deferring may result in improved performance in browsers that take advantage of the `defer` attribute. In HTML the `defer` attribute does not have a value; it simply must be present in the tag:

```
<script defer>
    // Any JavaScript code that does not call document.write( )
</script>
```

In XHTML, however, a value is required:

```
<script defer="defer"></script>
```

At the time of this writing, Internet Explorer is the only browser that uses the `defer` attribute. It does this when it is combined with the `src` attribute. It does not implement it quite correctly, however, because deferred scripts are always deferred until the end of the document, instead of simply being deferred until the next nondeferred script is encountered. This means that deferred scripts in IE are executed out of order and must not define any functions or set any variables that are required by the nondeferred scripts that follow.

13.2.5 The <noscript> Tag

HTML defines the `<noscript>` element to hold content that should be rendered only if JavaScript has been disabled in the browser. Ideally, you should to craft your web pages so that JavaScript serves as an enhancement only, and the pages "degrade gracefully" and still function without JavaScript. When this is not possible, however, you can use `<noscript>` to notify the users that JavaScript is required and possibly to provide a link to alternative content.

13.2.6 The </script> Tag

You may at some point find yourself writing a script that uses the `document.write()` method or `innerHTML` property to output some other script (typically into another window or frame). If you do this, you'll need to output a `</script>` tag to terminate the script you are creating. You must be careful, though: the HTML parser makes no attempt to understand your JavaScript code, and if it sees the string "</script>" in your code, even if it appears within quotes, it assumes that it has found the closing tag of the currently running script. To avoid this problem, simply break up the tag into pieces and write it out using an expression such as `"</" + "script>"`:

```
<script>
f1.document.write("<script>");
f1.document.write("document.write('<h2>This is the quoted script</h2>')");
f1.document.write("</" + "script>");
</script>
```

Alternatively, you can escape the `/` in `</script>` with a backslash:

```
f1.document.write("<\/script>");
```

In XHTML, scripts are enclosed in CDATA sections, and this problem with closing `</script>` tags does not occur.

13.2.7 Hiding Scripts from Old Browsers

When JavaScript was new, some browsers did not recognize the `<script>` tag and would therefore (correctly) render the content of this tag as text. The user visiting the web page would see JavaScript code formatted into big meaningless paragraphs and presented as web page content! The workaround to this problem was a simple hack that used HTML comments inside the script tag. JavaScript programmers habitually wrote their scripts like this:

```
<script language="JavaScript">
<!-- Begin HTML comment that hides the script
        // JavaScript statements go here
        //              .
        //              .
// End HTML comment that hides the script -->
</script>
```

Or, more compactly, like this:

```
<script><!--
// script body goes here
//--></script>
```

In order to make this work, client-side JavaScript tweaks the core JavaScript language slightly so that the character sequence `<!--` at the beginning of a script behaves just like `//`: it introduces a single-line comment.

The browsers that required this commenting hack are long gone, but you will probably still encounter the hack in existing web pages.

13.2.8 Nonstandard Script Attributes

Microsoft has defined two completely nonstandard attributes for the `<script>` tag that work only in Internet Explorer. The `event` and `for` attributes allow you to define event handlers using the `<script>` tag. The `event` attribute specifies the name of the event to be handled, and the `for` attribute specifies the name or ID of the element for which the event is to be handled. The content of the script is executed when the specified event occurs on the specified element.

These attributes work only in IE, and their functionality can easily be achieved in other ways. You should never use them; they are mentioned here only so that you will know what they are if you encounter them in existing web pages.

13.3 Event Handlers in HTML

JavaScript code in a script is executed once: when the HTML file that contains it is read into the web browser. A program that uses only this sort of static script cannot dynamically respond to the user. More dynamic programs define event handlers that are automatically invoked by the web browser when certain events occur—for example, when the user clicks on a button within a form. Because events in client-side JavaScript originate from HTML objects (such as buttons), event handlers can be defined as attributes of those objects. For example, to define an event handler that is invoked when the user clicks on a checkbox in a form, you specify the handler code as an attribute of the HTML tag that defines the checkbox:

```
<input type="checkbox" name="options" value="giftwrap"
       onclick="giftwrap = this.checked;"
>
```

What's of interest here is the `onclick` attribute. The string value of the `onclick` attribute may contain one or more JavaScript statements. If there is more than one statement, the statements must be separated from each other with semicolons. When the specified event—in this case, a click—occurs on the checkbox, the JavaScript code within the string is executed.

While you can include any number of JavaScript statements within an event-handler definition, a common technique is to simply use event-handler attributes to invoke functions that are defined elsewhere within `<script>` tags. This keeps most of your actual JavaScript code within scripts and reduces the need to mingle JavaScript and HTML.

Note that HTML event-handler attributes are not the only way to define JavaScript event handlers. Chapter 17 shows that it is possible to specify JavaScript event handlers for HTML elements using JavaScript code in a `<script>` tag. Some JavaScript developers argue that HTML event-handler attributes should never be used—that truly unobtrusive JavaScript requires a complete separation of content from behavior. According to this style of JavaScript coding, all JavaScript code should be placed in external files, referenced from HTML with the src attribute of `<script>` tags. This external JavaScript code can define whatever event handlers it needs when it runs.

Events and event handlers are covered in much more detail in Chapter 17, but you'll see them used in a variety of examples before then. Chapter 17 includes a comprehensive list of event handlers, but these are the most common:

`onclick`
: This handler is supported by all button-like form elements, as well as `<a>` and `<area>` tags. It is triggered when the user clicks on the element. If an `onclick` handler returns `false`, the browser does not perform any default action associated with the button or link; for example, it doesn't follow a hyperlink (for an `<a>` tag) or submit a form (for a **Submit** button).

`onmousedown`, `onmouseup`
: These two event handlers are a lot like `onclick`, but they are triggered separately when the user presses and releases a mouse button, respectively. Most document elements support these handlers.

`onmouseover`, `onmouseout`
: These two event handlers are triggered when the mouse pointer moves over or out of a document element, respectively.

`onchange`
: This event handler is supported by the `<input>`, `<select>`, and `<textarea>` elements. It is triggered when the user changes the value displayed by the element and then tabs or otherwise moves focus out of the element.

`onload`
: This event handler may appear on the `<body>` tag and is triggered when the document and its external content (such as images) are fully loaded. The `onload` handler is often used to trigger code that manipulates the document content because it indicates that the document has reached a stable state and is safe to modify.

For a realistic example of the use of event handlers, take another look at the interactive loan-payment script in Example 1-3. The HTML form in this example contains a number of event-handler attributes. The body of these handlers is simple: they simply call the `calculate( )` function defined elsewhere within a `<script>`.

13.4 JavaScript in URLs

Another way that JavaScript code can be included on the client side is in a URL following the `javascript:` pseudoprotocol specifier. This special protocol type specifies that the body of the URL is an arbitrary string of JavaScript code to be run by the JavaScript interpreter. It is treated as a single line of code, which means that statements must be separated by semicolons and that `/* */` comments must be used in place of `//` comments. A JavaScript URL might look like this:

```
javascript:var now = new Date( ); "<h1>The time is:</h1>" + now;
```

When the browser loads one of these JavaScript URLs, it executes the JavaScript code contained in the URL and uses the value of the last JavaScript statement or expression, converted to a string, as the contents of the new document to display. This string value may contain HTML tags and is formatted and displayed just like any other document loaded into the browser.

JavaScript URLs may also contain JavaScript statements that perform actions but return no value. For example:

```
javascript:alert("Hello World!")
```

When this sort of URL is loaded, the browser executes the JavaScript code, but because there is no value to display as the new document, it does not modify the currently displayed document.

Often, you'll want to use a JavaScript URL to execute some JavaScript code without altering the currently displayed document. To do this, be sure that the last statement in the URL has no return value. One way to ensure this is to use the `void` operator to explicitly specify an undefined return value. Simply use the statement `void 0;` at the end of your JavaScript URL. For example, here is a URL that opens a new, blank browser window without altering the contents of the current window:

```
javascript:window.open("about:blank"); void 0;
```

Without the `void` operator in this URL, the return value of the `Window.open()` method call would be converted to a string and displayed, and the current document would be overwritten by a document that appears something like this:

```
[object Window]
```

You can use a JavaScript URL anywhere you'd use a regular URL. One handy way to use this syntax is to type it directly into the **Location** field of your browser, where you can test arbitrary JavaScript code without having to open your editor and create an HTML file containing the code.

The `javascript:` pseudoprotocol can be used with HTML attributes whose value should be a URL. The `href` attribute of a hyperlink is one such case. When the user clicks on such a link, the specified JavaScript code is executed. In this context, the JavaScript URL is essentially a substitute for an `onclick` event handler. (Note that using either an `onclick` handler or a JavaScript URL with an HTML link is normally a bad design choice; use a button instead, and reserve links for loading new documents.) Similarly, a JavaScript URL can be used as the value of the `action` attribute of a `<form>` tag so that the JavaScript code in the URL is executed when the user submits the form.

JavaScript URLs can also be passed to methods, such as `Window.open()` (see Chapter 14), that expect URL arguments.

13.4.1 Bookmarklets

One particularly important use of `javascript:` URLs is in bookmarks, where they form useful mini-JavaScript programs, or *bookmarklets*, that can be easily launched from a menu or toolbar of bookmarks. The following HTML snippet includes an <a> tag with a `javascript:` URL as the value of its `href` attribute. Clicking the link opens a simple JavaScript expression evaluator that allows you to evaluate expressions and execute statements in the context of the page:

```
<a href='javascript:
var e = "", r = ""; /* Expression to evaluate and the result */
do {
    /* Display expression and result and ask for a new expression */
    e = prompt("Expression: " + e + "\n" + r + "\n", e);
    try { r = "Result: " + eval(e); } /* Try to evaluate the expression */
    catch(ex) { r = ex; }             /* Or remember the error instead  */
} while(e);  /* Continue until no expression entered or Cancel clicked */
void 0;      /* This prevents the current document from being overwritten */
'>
JavaScript Evaluator
</a>
```

Note that even though this JavaScript URL is written across multiple lines, the HTML parser treats it as a single line, and single-line // comments will not work in it. Here's what the link looks like with comments and whitespace stripped out:

```
<a href='javascript:var e="",r="";do{e=prompt("Expression: "+e+"\n"+r+"\n",e);
try{r="Result: "+eval(e);}catch(ex){r=ex;}}while(e);void 0;'>JS Evaluator</a>
```

A link like this is useful when hardcoded into a page that you are developing but becomes much more useful when stored as a bookmark that you can run on any page. Typically you can store a bookmark by right-clicking on the link and selecting **Bookmark This Link** or some similar option. In Firefox, you can simply drag the link to your bookmarks toolbar.

The client-side JavaScript techniques covered in this book are all applicable to the creation of bookmarklets, but bookmarklets themselves are not covered in any detail. If you are intrigued by the possibilities of these little programs, try an Internet search for "bookmarklets". You will find a number of sites that host many interesting and useful bookmarklets.

13.5 Execution of JavaScript Programs

The previous sections discussed the mechanics of integrating JavaScript code into an HTML file. Now the following sections discuss exactly how and when that integrated JavaScript code is executed by the JavaScript interpreter.

13.5.1 Executing Scripts

JavaScript statements that appear between `<script>` and `</script>` tags are executed in the order that they appear in the script. When a file has more than one script, the scripts are executed in the order in which they appear (with the exception of scripts with the `defer` attribute, which IE executes out of order). The JavaScript code in `<script>` tags is executed as part of the document loading and parsing process.

Any `<script>` element that does not have a `defer` attribute may call the `document.write( )` method (described in detail in Chapter 15). The text passed to this method is inserted into the document at the location of the scripts. When the script is finished executing, the HTML parser resumes parsing the document, starting with any text output by the script.

Scripts can appear in the `<head>` or the `<body>` of an HTML document. Scripts in the `<head>` typically define functions to be called by other code. They may also declare and initialize variables that other code will use. It is common for scripts in the `<head>` of a document to define a single function and then register that function as an `onload` event handler for later execution. It is legal, but uncommon, to call `document.write( )` in the `<head>` of a document.

Scripts in the `<body>` of a document can do everything that scripts in the `<head>` can do. It is more common to see calls to `document.write( )` in these scripts, however. Scripts in the `<body>` of a document may also (using techniques described in Chapter 15) access and manipulate document elements and document content that appear before the script. As described later in this chapter, however, document elements are not guaranteed to be available and stable when the scripts in the `<body>` are executed. If a script simply defines functions and variables to be used later and does not call `document.write( )` or otherwise attempt to modify document content, convention dictates that it should appear in the `<head>` of the document instead of the `<body>`.

As previously mentioned, IE executes scripts with the `defer` attribute out of order. These scripts are run after all nondeferred scripts and after the document is fully parsed, but before the onload event handler is triggered.

13.5.2 The onload Event Handler

After the document is parsed, all scripts have run, and all auxiliary content (such as images) has loaded, the browser fires the onload event and runs any JavaScript code that has been registered with the Window object as an `onload` event handler. An `onload` handler can be registered by setting the `onload` attribute of the `<body>` tag. It is also possible (using techniques shown in Chapter 17) for separate modules of JavaScript code to register their own onload event handlers. When more than one `onload` handler is registered, the browser invokes all handlers, but there is no guarantee about the order in which they are invoked.

When the `onload` handler is triggered, the document is fully loaded and parsed, and any document element can be manipulated by JavaScript code. For this reason, JavaScript modules that modify document content typically contain a function to perform the modification and event-registration code that arranges for the function to be invoked when the document is fully loaded.

Because `onload` event handlers are invoked after document parsing is complete, they must not call `document.write( )`. Instead of appending to the current document, any such call would instead begin a new document and overwrite the current document before the user even had a chance to view it.

13.5.3 Event Handlers and JavaScript URLs

When document loading and parsing ends, the `onload` handler is triggered, and JavaScript execution enters its event-driven phase. During this phase, event handlers are executed asynchronously in response to user input such as mouse motion, mouse clicks, and key presses. JavaScript URLs may be invoked asynchronously during this phase as well, if, for example, the user clicks on a link whose `href` attribute uses the `javascript:` pseudoprotocol.

<script> elements are typically used to define functions, and event handlers are typically used to invoke those functions in response to user input. Event handlers can define functions, of course, but this is an uncommon (and not very useful) thing to do.

If an event handler calls `document.write()` on the document of which it is a part, it will overwrite that document and begin a new one. This is almost never what is intended, and, as a rule of thumb, event handlers should never call this method. Nor should they call functions that call this method. The exception, however, is in multiwindow applications in which an event handler in one window invokes the `write()` method of the document of a different window. (See Section 14.8 for more on multiwindow JavaScript applications.)

13.5.4 The onunload Event Handler

When the user navigates away from a web page, the browser triggers the `onunload` event handler, giving the JavaScript code on that page one final chance to run. You can define an `onunload` handler by setting the `onunload` attribute of the `<body>` tag or with other event-handler registration techniques described in Chapter 17.

The `onunload` event enables you to undo the effects of your `onload` handler or other scripts in your web page. For example, if your application opens up a secondary browser window, the `onunload` handler provides an opportunity to close that window when the user leaves your main page. The `onunload` handler should not run any time-consuming operation, nor should it pop up a dialog box. It exists simply to perform a quick cleanup operation; running it should not slow down or impede the user's transition to a new page.

13.5.5 The Window Object as Execution Context

All scripts, event handlers, and JavaScript URLs in a document share the same Window object as their global object. JavaScript variables and functions are nothing more than properties of the global object. This means that a function declared in one `<script>` can be invoked by the code in any subsequent `<script>`.

Since the onload event is not triggered until after all scripts have executed, every `onload` event handler has access to all functions defined and variables declared by all scripts in the document.

Whenever a new document is loaded into a window, the Window object for that window is restored to its default state: any properties and functions defined by a script in the previous document are deleted, and any of the standard system properties that may have been altered or overwritten are restored. Every document begins with a clean slate. Your scripts can rely on this; they will not inherit a corrupted environment from the previous document. This also means that any variables and functions your scripts define persist only until the document is replaced with a new one.

The properties of a Window object have the same lifetime as the document that contains the JavaScript code that defined those properties. A Window object itself has a longer lifetime; it exists as long as the window it represents exists. A reference to a Window object remains valid regardless of how many web pages the window loads and unloads. This is relevant only for web applications that use multiple windows or frames. In this case, JavaScript code in one window or frame may maintain a reference to another window or frame. That reference remains valid even if the other window or frame loads a new document.

13.5.6 Client-Side JavaScript Threading Model

The core JavaScript language does not contain any threading mechanism, and client-side JavaScript does not add any. Client-side JavaScript is (or behaves as if it is) single-threaded. Document parsing stops while scripts are loaded and executed, and web browsers stop responding to user input while event handlers are being executed.

Single-threaded execution makes for much simpler scripting: you can write code with the assurance that two event handlers will never run at the same time. You can manipulate document content knowing that no other thread is attempting to modify it at the same time.

Single-threaded execution also places a burden on JavaScript programmers: it means that JavaScript scripts and event handlers must not run for too long. If a script performs a computationally intensive task, it will introduce a delay into document loading, and the user will not see the document content until the script completes. If an event handler performs a computationally intensive task, the browser may become nonresponsive, possibly causing the user to think that it has crashed.*

If your application must perform enough computation to cause a noticeable delay, you should allow the document to load fully before performing that computation, and you should be sure to notify the user that computation is underway and that the browser is not hung. If it is possible to break your computation down into discrete subtasks, you can use methods such as `setTimeout()` and `setInterval()` (see Chapter 14) to run the subtasks in the background while updating a progress indicator that displays feedback to the user.

13.5.7 Manipulating the Document During Loading

While a document is being loaded and parsed, JavaScript code in a `<script>` element can insert content into the document with `document.write()`. Other kinds of document manipulation, using DOM scripting techniques shown in Chapter 15, may or may not be allowed in `<script>` tags.

* Some browsers, such as Firefox, guard against denial-of-service attacks and accidental infinite loops, and prompt the user if a script or event handler takes too long to run. This gives the user the chance to abort a runaway script.

Most browsers seem to allow scripts to manipulate any document elements that appear before the `<script>` tag. Some JavaScript coders do this routinely. However, no standard requires it to work, and there is a persistent, if vague, belief among some experienced JavaScript coders that placing document manipulation code within `<script>` tags can cause problems (perhaps only occasionally, only with some browsers, or only when a document is reloaded or revisited with the browser's **Back** button).

The only consensus that exists in this gray area is that it is safe to manipulate the document once the onload event has been triggered, and this is what most JavaScript applications do: they use the `onload` handler to trigger all document modifications. I present a utility routine for registering `onload` event handlers in Example 17-7.

In documents that contain large images or many images, the main document may be parsed well before the images are loaded and the onload event is triggered. In this case, you might want to begin manipulating the document before the onload event. One technique (whose safety is debated) is to place the manipulation code at the end of the document. An IE-specific technique is to put the document manipulation code in a `<script>` that has both `defer` and `src` attributes. A Firefox-specific technique is to make the document-manipulation code an event handler for the undocumented DOMContentLoaded event, which is fired when the document is parsed but before external objects, such as images, are fully loaded.

Another gray area in the JavaScript execution model is the question of whether event handlers can be invoked before the document is fully loaded. Our discussion of the JavaScript execution model has so far concluded that all event handlers are always triggered after all scripts have been executed. While this typically happens, it is not required by any standard. If a document is very long or is being loaded over a slow network connection, the browser might partially render the document and allow the user to begin interacting with it (and triggering event handlers) before all scripts and `onload` handlers have run. If such an event handler invokes a function that is not yet defined, it will fail. (This is one reason to define all functions in scripts in the `<head>` of a document.) And if such an event handler attempts to manipulate a part of the document that has not yet been parsed, it will fail. This scenario is uncommon in practice, and it is not usually worth the extra coding effort required to aggressively protect against it.

13.6 Client-Side Compatibility

The web browser is a universal platform for hosting applications, and JavaScript is the language in which those applications are developed. Fortunately, the JavaScript language is standardized and well-supported: all modern web browsers support ECMAScript v3. The same can not be said for the platform itself. All web browsers display HTML, of course, but they differ in their support for other standards such as

CSS (Cascading Style Sheets) and the DOM. And although all modern browsers include a compliant JavaScript interpreter, they differ in the APIs they make available to client-side JavaScript code.

Compatibility issues are simply an unpleasant fact of life for client-side JavaScript programmers. The JavaScript code you write and deploy may be run in various versions of various browsers running on various operating systems. Consider the permutations of popular operating systems and browsers: Internet Explorer on Windows and Mac OS;* Firefox on Windows, Mac OS, and Linux; Safari on Mac OS; and Opera on Windows, Mac OS, and Linux. If you want to support the current version of each browser plus the previous two versions, multiply these nine browser/OS pairs by three, for a total of 27 browser/version/OS combinations. The only way to be absolutely sure that your web application runs on all 27 combinations is to test it in each. This is a daunting task, and in practice, the testing is often done by the users after the application is deployed!

Before you reach the testing phase of application development, you must write the code. When programming in JavaScript, knowledge of the incompatibilities among browsers is crucial for creating compatible code. Unfortunately, producing a definitive listing of all known vendor, version, and platform incompatibilities would be an enormous task. It is beyond the scope and mission of this book, and to my knowledge, no comprehensive client-side JavaScript test suite has ever been developed. You can find browser compatibility information online, and here are two sites that I have found useful:

http://www.quirksmode.org/dom/
: This is freelance web developer Peter-Paul Koch's web site. His DOM compatibility tables show the compatibility of various browsers with the W3C DOM.

http://webdevout.net/browser_support.php
: This site by David Hammond is similar to *quirksmode.org*, but its compatibility tables are more comprehensive and (at the time of this writing) somewhat more up-to-date. In addition to DOM compatibility, it also rates browser compliance with the HTML, CSS, and ECMAScript standards.

Awareness of incompatibilities is only the first step, of course. The subsections that follow demonstrate techniques you can use to work around the incompatibilities you encounter.

13.6.1 The History of Incompatibility

Client-side JavaScript programming has always been about coping with incompatibility. Knowing the history provides some useful context. The early days of web programming were marked by the "browser wars" between Netscape and Microsoft.

* IE for Mac is being phased out, which is a blessing because it is substantially different from IE for Windows.

This was an intense burst of development, in often incompatible directions, of the browser environment and client-side JavaScript APIs. Incompatibility problems were at their worst at this point, and some web sites simply gave up and told their visitors which browser they needed to use to access the site.

The browser wars ended, with Microsoft holding a dominant market share, and web standards, such as the DOM and CSS, started to take hold. A period of stability (or stagnation) followed while the Netscape browser slowly morphed into the Firefox browser and Microsoft made a few incremental improvements to its browser. Standards support in both browsers was good, or at least good enough for compatible web applications to be written.

At the time of this writing, we seem to be at the start of another burst of browser innovation. For example, all major browsers now support scripted HTTP requests, which form the cornerstone of the new Ajax web application architecture (see Chapter 20). Microsoft is working on Internet Explorer 7, which will address a number of long-standing security and CSS compatibility issues. IE 7 will have many user-visible changes but will not, apparently, break new ground for web developers. Other browsers are breaking new ground, however. For example, Safari and Firefox support a `<canvas>` tag for scripted client-side graphics (see Chapter 22). A consortium of browser vendors (with the notable absence of Microsoft) known as WHATWG (*whatwg.org*) is working to standardize the `<canvas>` tag and many other extensions to HTML and the DOM.

13.6.2 A Word about "Modern Browsers"

Client-side JavaScript is a moving target, especially if we're indeed entering a period of rapid evolution. For this reason, I shy away in this book from making narrow statements about particular versions of particular browsers. Any such claims are likely to be outdated before I can write a new edition of the book. A printed book like this simply cannot be updated as often as necessary to provide a useful guide to the compatibility issues that affect the current crop of browsers.

You'll find, therefore, that I often hedge my statements with purposely vague language like "all modern browsers" (or sometimes "all modern browsers except IE"). At the time of this writing, the loose set of "modern browsers" includes: Firefox 1.0, Firefox 1.5, IE 5.5, IE 6.0, Safari 2.0, Opera 8, and Opera 8.5. This is not a guarantee that every statement in this book about "modern browsers" is true for each of these specific browsers. However, it allows you to know what browsers were current technology when this book was written.

13.6.3 Feature Testing

Feature testing (sometimes called *capability testing*) is a powerful technique for coping with incompatibilities. If you want to use a feature or capability that may not be supported by all browsers, include code in your script that tests to see whether that feature is supported. If the desired feature is not supported on the current platform, either do not use it on that platform or provide alternative code that works on all platforms.

You'll see feature testing again and again in the chapters that follow. In Chapter 17, for example, there is code that looks like this:

```
if (element.addEventListener) { // Test for this W3C method before using it
    element.addEventListener("keydown", handler, false);
    element.addEventListener("keypress", handler, false);
}
else if (element.attachEvent) { // Test for this IE method before using it
    element.attachEvent("onkeydown", handler);
    element.attachEvent("onkeypress", handler);
}
else {  // Otherwise, fall back on a universally supported technique
    element.onkeydown = element.onkeypress = handler;
}
```

Chapter 20 describes yet another approach to feature testing: keep trying alternatives until you find one that does not throw an exception! And, when you find an alternative that works, remember it for future use. Here is a preview of code from Example 20-1:

```
// This is a list of XMLHttpRequest creation functions to try
HTTP._factories = [
    function( ) { return new XMLHttpRequest( ); },
    function( ) { return new ActiveXObject("Msxml2.XMLHTTP"); },
    function( ) { return new ActiveXObject("Microsoft.XMLHTTP"); }
];

// When we find a factory that works, store it here
HTTP._factory = null;

// Create and return a new XMLHttpRequest object.
//
// The first time we're called, try the list of factory functions until
// we find one that returns a nonnull value and does not throw an
// exception. Once we find a working factory, remember it for later use.
HTTP.newRequest = function( ) { /* fuction body omitted */ }
```

A common, but outdated, example of feature testing that you may still encounter in existing code is used to determine which DOM a browser supports. It often occurs in DHTML code and usually looks something like this:

```
if (document.getElementById) {  // If the W3C DOM API is supported,
    // do our DHTML using the W3C DOM API
}
else if (document.all) {        // If the IE 4 API is supported,
    // do our DHTML using the IE 4 API
}
else if (document.layers) {     // If the Netscape 4 API is supported,
        // do the DHTML effect (as best we can) using the Netscape 4 API
    }
    else {                          // Otherwise, DHTML is not supported,
        // so provide a static alternative to DHTML
    }
```

Code like this is outdated because almost all browsers deployed today support the W3C DOM and its `document.getElementById()` function.

The important thing about the feature-testing technique is that it results in code that is not tied to a specific list of browser vendors or browser version numbers. It works with the set of browsers that exist today and should continue to work with future browsers, whatever feature sets they implement. Note, however, that it requires browser vendors not to define a property or method unless that property or method is fully functional. If Microsoft were to define an `addEventHandler()` method that only partially implemented the W3C specification, it would break a lot of code that uses feature testing before calling `addEventHandler()`.

The `document.all` property shown in this example deserves a special mention here. The `document.all[]` array was introduced by Microsoft in IE 4. It allowed JavaScript code to refer to all elements of a document and ushered in a new era of client-side programming. It was never standardized and was superseded by `document.getElementById()`. It is still used in existing code and has often been used (incorrectly) to determine whether a script is running in IE with code like this:

```
    if (document.all) {
        // We're running in IE
    }
    else {
        // We're in some other browser
    }
```

Because there is still a lot of extant code that uses `document.all`, the Firefox browser has added support for it so that Firefox can work with sites that were previously IE-dependent. Because the presence of the `all` property is often used for browser detection, Firefox pretends that it does not support the property. So even though Firefox does support `document.all`, the `if` statement in the following script behaves as if the `all` property does not exist, and the script displays a dialog box containing the text "Firefox":

```
    if (document.all) alert("IE"); else alert("Firefox");
```

This example illustrates that the feature-testing approach does not work if the browser actively lies to you! It also shows that web developers are not the only ones plagued by compatibility issues. Browser vendors must also go through contortions for compatibility.

13.6.4 Browser Testing

Feature testing is well suited to checking for support of large functional areas. You can use it to determine whether a browser supports the W3C event-handling model or the IE event-handling model, for example. On the other hand, sometimes you may need to work around individual bugs or quirks in a particular browser, and there may be no easy way to test for the existence of the bug. In this case, you need to create a platform-specific workaround that is tied to a particular browser vendor, version, or operating system (or some combination of the three).

The way to do this in client-side JavaScript is with the Navigator object, which you'll learn about in Chapter 14. Code that determines the vendor and version of the current browser is often called a *browser sniffer* or a *client sniffer*. A simple example is shown in Example 14-3. Client sniffing was a common client-side programming technique in the early days of the Web when the Netscape and IE platforms were incompatible and diverging. Now that the compatibility situation has stabilized, client sniffing has fallen out of favor and should be used only when absolutely necessary.

Note that client sniffing can be done on the server side as well, with the web server choosing what JavaScript code to send based on how the browser identifies itself in its `User-Agent` header.

13.6.5 Conditional Comments in Internet Explorer

In practice, you'll find that many of the incompatibilities in client-side JavaScript programming turn out to be IE-specific. That is, you must write code in one way for IE and in another way for all other browsers. Although you should normally avoid browser-specific extensions that are not likely to be standardized, IE supports conditional comments in both HTML and JavaScript code that can be useful.

Here is what conditional comments in HTML look like. Notice the tricks played with the closing delimiter of HTML comments:

```
<!--[if IE]>
This content is actually inside an HTML comment.
It will only be displayed in IE.
<![endif]-->

<!--[if gte IE 6]>
This content will only be displayed by IE 6 and later.
<![endif]-->

<!--[if !IE]> <-->
This is normal HTML content, but IE will not display it
because of the comment above and the comment below.
<!--> <![endif]-->

This is normal content, displayed by all browsers.
```

Conditional comments are also supported by IE's JavaScript interpreter, and C and C++ programmers may find them similar to the `#ifdef/#endif` functionality of the C preprocessor. A JavaScript conditional comment in IE begins with the text `/*@cc_on` and ends with the text `@*/`. (The cc in `cc_on` stands for conditional compilation.) The following conditional comment includes code that is executed only in IE:

```
/*@cc_on
  @if (@_jscript)

    // This code is inside a JS comment but is executed in IE.
    alert("In IE");

  @end
  @*/
```

Inside a conditional comment, the keywords `@if`, `@else`, and `@end` delimit the code that is to be conditionally executed by IE's JavaScript interpreter. Most of the time, you need only the simple conditional shown above: `@if (@_jscript)`. JScript is Microsoft's name for its JavaScript interpreter, and the `@_jscript` variable is always `true` in IE.

With clever interleaving of conditional comments and regular JavaScript comments, you can set up one block of code to run in IE and a different block to run in all other browsers:

```
/*@cc_on
  @if (@_jscript)
    // This code is inside a conditional comment, which is also a
    // regular JavaScript comment. IE runs it but other browsers ignore it.
    alert('You are using Internet Explorer);
  @else*/
    // This code is no longer inside a JavaScript comment, but is still
    // inside the IE conditional comment.  This means that all browsers
    // except IE will run this code.
    alert('You are not using Internet Explorer');
/*@end
  @*/
```

Conditional comments, in both their HTML and JavaScript forms, are completely nonstandard. They are sometimes a useful way to achieve compatibility with IE, however.

13.7 Accessibility

The Web is a wonderful tool for disseminating information, and JavaScript programs can enhance access to that information. JavaScript programmers must be careful, however: it is easy to write JavaScript code that inadvertently denies information to visitors with visual or physical handicaps.

Blind users may use a form of "assistive technology" known as a screen reader to convert written words to spoken words. Some screen readers are JavaScript-aware, and others work best when JavaScript is turned off. If you design a web site that requires JavaScript to display its information, you exclude the users of these screen readers. (And you have also excluded anyone who browses with a mobile device, such as a cell phone, that does not have JavaScript support, as well as anyone else who intentionally disables JavaScript in his browser.) The proper role of JavaScript is to enhance the presentation of information, not to take over the presentation of that information. A cardinal rule of JavaScript accessibility is to design your code so that the web page on which it is used will still function (at least in some form) with the JavaScript interpreter turned off.

Another important accessibility concern is for users who can use the keyboard but cannot use (or choose not to use) a pointing device such as a mouse. If you write JavaScript code that relies on mouse-specific events, you exclude users who do not use the mouse. Web browsers allow keyboard traversal and activation of a web page, and your JavaScript code should as well. And at the same time, you should not write code that requires keyboard input either, or you will exclude users who cannot use a keyboard as well as many users of tablet PCs and cell phone browsers. As shown in Chapter 17, JavaScript supports device-independent events, such as `onfocus` and `onchange`, as well as device-dependent events, such as `onmouseover` and `onmousedown`. For accessibility, you should favor the device-independent events whenever possible.

Creating accessible web pages is a nontrivial problem without clear-cut solutions. At the time of this writing, debate continues on how to best use JavaScript to foster, rather than degrade, accessibility. A full discussion of JavaScript and accessibility is beyond the scope of this book. An Internet search will yield a lot of information on this topic, much of it couched in the form of recommendations from authoritative sources. Keep in mind that both client-side JavaScript programming practices and assistive technologies are evolving, and accessibility guidelines do not always keep up.

13.8 JavaScript Security

Internet security is a broad and complex field. This section focuses on client-side JavaScript security issues.

13.8.1 What JavaScript Can't Do

The introduction of JavaScript interpreters into web browsers means that loading a web page can cause arbitrary JavaScript code to be executed on your computer. Secure web browsers—and commonly used modern browsers appear to be relatively secure—restrict scripts in various ways to prevent malicious code from reading confidential data, altering your data, or compromising your privacy.

JavaScript's first line of defense against malicious code is that the language simply does not support certain capabilities. For example, client-side JavaScript does not provide any way to read, write, or delete files or directories on the client computer. With no File object and no file-access functions, a JavaScript program cannot delete a user's data or plant viruses on a user's system.

The second line of defense is that JavaScript imposes restrictions on certain features that it does support. For example, client-side JavaScript can script the HTTP protocol to exchange data with web servers, and it can even download data from FTP and other servers. But JavaScript does not provide general networking primitives and cannot open a socket to, or accept a connection from, another host.

The following list includes other features that may be restricted. Note that this is not a definitive list. Different browsers have different restrictions, and many of these restrictions may be user-configurable:

- A JavaScript program can open new browser windows, but, to prevent pop-up abuse by advertisers, many browsers restrict this feature so that it can happen only in response to a user-initiated event such as a mouse click.
- A JavaScript program can close browser windows that it opened itself, but it is not allowed to close other windows without user confirmation. This prevents malicious scripts from calling `self.close()` to close the user's browsing window, thereby causing the program to exit.
- A JavaScript program cannot obscure the destination of a link by setting the status line text when the mouse moves over the link. (It was common in the past to provide additional information about a link in the status line. Abuse by phishing scams has caused many browser vendors to disable this capability.)
- A script cannot open a window that is too small (typically smaller than 100 pixels on a side) or shrink a window too small. Similarly, a script cannot move a window off the screen or create a window that is larger than the screen. This prevents scripts from opening windows that the user cannot see or could easily overlook; such windows could contain scripts that keep running after the user thinks they have stopped. Also, a script may not create a browser window without a titlebar or status line because such a window could spoof an operating dialog box and trick the user into entering a sensitive password, for example.
- The `value` property of HTML FileUpload elements cannot be set. If this property could be set, a script could set it to any desired filename and cause the form to upload the contents of any specified file (such as a password file) to the server.
- A script cannot read the content of documents loaded from different servers than the document that contains the script. Similarly, a script cannot register event listeners on documents from different servers. This prevents scripts from snooping on the user's input (such as the keystrokes that constitute a password entry) to other pages. This restriction is known as the *same-origin policy* and is described in more detail in the next section.

13.8.2 The Same-Origin Policy

The *same-origin policy* is a sweeping security restriction on what web content JavaScript code can interact with. It typically comes into play when a web page uses multiple frames, includes `<iframe>` tags, or opens other browser windows. In this case, the same-origin policy governs the interactions of JavaScript code in one window or frame with other windows and frames. Specifically, a script can read only the properties of windows and documents that have the same origin as the document that contains the script (see Section 14.8 to learn how to use JavaScript with multiple windows and frames).

The same-origin policy also comes up when scripting HTTP with the XMLHttpRequest object. This object allows client-side JavaScript code to make arbitrary HTTP requests but only to the web server from which the containing document was loaded (see Chapter 20 for more on the XMLHttpRequest object).

The *origin* of a document is defined as the protocol, host, and port of the URL from which the document was loaded. Documents loaded from different web servers have different origins. Documents loaded through different ports of the same host have different origins. And a document loaded with the `http:` protocol has a different origin than one loaded with the `https:` protocol, even if they come from the same web server.

It is important to understand that the origin of the script itself is not relevant to the same-origin policy: what matters is the origin of the document in which the script is embedded. Suppose, for example, that a script from domain A is included (using the `src` property of the `<script>` tag) in a web page in domain B. That script has full access to the content of the document that contains it. If the script opens a new window and loads a second document from domain B, the script also has full access to the content of that second document. But if the script opens a third window and loads a document from domain C (or even from domain A) into it, the same-origin policy comes into effect and prevents the script from accessing this document.

The same-origin policy does not actually apply to all properties of all objects in a window from a different origin. But it does apply to many of them, and, in particular, it applies to practically all the properties of the Document object (see Chapter 15). Furthermore, different browser vendors implement this security policy somewhat differently. (For example, Firefox 1.0 allows a script to call `history.back()` on different-origin windows, but IE 6 does not.) For all intents and purposes, therefore, you should consider any window that contains a document from another server to be off-limits to your scripts. If your script opened the window, your script can close it, but it cannot "look inside" the window in any way.

The same-origin policy is necessary to prevent scripts from stealing proprietary information. Without this restriction, a malicious script (loaded through a firewall into a browser on a secure corporate intranet) might open an empty window, hoping to trick the user into using that window to browse files on the intranet. The malicious script would then read the content of that window and send it back to its own server. The same-origin policy prevents this kind of behavior.

In some circumstances, the same-origin policy is too restrictive. It poses particular problems for large web sites that use more than one server. For example, a script from *home.example.com* might legitimately want to read properties of a document loaded from *developer.example.com*, or scripts from *orders.example.com* might need to read properties from documents on *catalog.example.com*. To support large web sites of this sort, you can use the `domain` property of the Document object. By default, the `domain` property contains the hostname of the server from which the document was loaded. You can set this property, but only to a string that is a valid domain suffix of itself. Thus, if `domain` is originally the string "home.example.com", you can set it to the string "example.com", but not to "home.example" or "ample.com". Furthermore, the `domain` value must have at least one dot in it; you cannot set it to "com" or any other top-level domain.

If two windows (or frames) contain scripts that set `domain` to the same value, the same-origin policy is relaxed for these two windows, and each window can interact with the other. For example, cooperating scripts in documents loaded from *orders.example.com* and *catalog.example.com* might set their `document.domain` properties to "example.com", thereby making the documents appear to have the same origin and enabling each document to read properties of the other.

13.8.3 Scripting Plug-ins and ActiveX Controls

Although the core JavaScript language and the basic client-side object model lack the filesystem and networking features that most malicious code requires, the situation is not quite as simple as it appears. In many web browsers, JavaScript is used as a "script engine" for other software components, such as ActiveX controls in Internet Explorer and plug-ins in other browsers. This exposes important and powerful features to client-side scripts. You'll see examples in Chapter 20, where an ActiveX control is used for scripting HTTP, and in Chapters 19 and 22, where the Java and Flash plug-ins are used for persistence and advanced client-side graphics.

There are security implications to being able to script ActiveX controls and plug-ins. Java applets, for example, have access to low-level networking capabilities. The Java security "sandbox" prevents applets from communicating with any server other than the one from which they were loaded, so this does not open a security hole. But it exposes the basic problem: if plug-ins are scriptable, you must trust not just the web browser's security architecture, but also the plug-in's security architecture. In practice, the Java and Flash plug-ins seem to have robust security and do not appear to introduce security issues into client-side JavaScript. ActiveX scripting has had a more checkered past, however. The IE browser has access to a variety of scriptable ActiveX controls that are part of the Windows operating system, and in the past some of these scriptable controls have included exploitable security holes. At the time of this writing, however, these problems appear to have been resolved.

13.8.4 Cross-Site Scripting

Cross-site scripting, or XSS, is a term for a category of security issues in which an attacker injects HTML tags or scripts into a target web site. Defending against XSS attacks is typically the job of server-side web developers. However, client-side JavaScript programmers must also be aware of, and defend against, cross-site scripting.

A web page is vulnerable to cross-site scripting if it dynamically generates document content and bases that content on user-submitted data without first "sanitizing" that data by removing any embedded HTML tags from it. As a trivial example, consider the following web page that uses JavaScript to greet the user by name:

```
<script>
var name = decodeURIComponent(window.location.search.substring(6)) || "";
document.write("Hello " + name);
</script>
```

This two-line script uses `window.location.search` to obtain the portion of its own URL that begins with ?. It uses `document.write()` to add dynamically generated content to the document. This page is intended to be invoked with a URL like this:

```
http://www.example.com/greet.html?name=David
```

When used like this, it displays the text "Hello David". But consider what happens when it is invoked with this URL:

```
http://www.example.com/greet.html?name=%3Cscript%3Ealert('David')%3C/script%3E
```

With this URL, the script dynamically generates another script (`%3C` and `%3E` are codes for angle brackets)! In this case, the injected script simply displays a dialog box, which is relatively benign. But consider this case:

```
http://siteA/greet.html?name=%3Cscript src=siteB/evil.js%3E%3C/script%3E
```

Cross-site scripting attacks are so called because more than one site is involved. Site B (or even site C) includes a specially crafted link (like the one above) to site A that injects a script from site B. The script *evil.js* is hosted by the evil site B, but it is now embedded in site A, and can do absolutely anything it wants with site A's content. It might deface the page or cause it to malfunction (such as by initiating one of the denial-of-service attacks described in the next section). This would be bad for site A's customer relations. More dangerously, the malicious script can read cookies stored by site A (perhaps account numbers or other personally identifying information) and send that data back to site B. The injected script can even track the user's keystrokes and send that data back to site B.

In general, the way to prevent XSS attacks is to remove HTML tags from any untrusted data before using it to create dynamic document content. You can fix the *greet.html* file shown earlier by adding this line of code to remove the angle brackets around `<script>` tags.

```
name = name.replace(/</g, "&lt;").replace(/>/g, "&gt;");
```

Cross-site scripting enables a pernicious vulnerability whose roots go deep into the architecture of the Web. It is worth understanding this vulnerability in depth, but further discussion is beyond the scope of this book. There are many online resources to help you defend against cross-site scripting. One important primary source is the original CERT Advisory about this problem: *http://www.cert.org/advisories/CA-2000-02.html*.

13.8.5 Denial-of-Service Attacks

The same-origin policy and other security restrictions described here do a good job of preventing malicious code from damaging your data or compromising your privacy. They do not protect against brute-force denial-of-service attacks, however. If you visit a malicious web site with JavaScript enabled, that site can tie up your browser with an infinite loop of `alert()` dialog boxes, forcing you to use, for example, the Unix `kill` command or the Windows Task Manager to shut your browser down.

A malicious site can also attempt to tie up your CPU with an infinite loop or meaningless computation. Some browsers (such as Firefox) detect long-running scripts and give the user the option to stop them. This defends against accidental infinite loops, but malicious code can use techniques such as the `window.setInterval()` command to avoid being shut down. A similar attack ties up your system by allocating lots of memory.

There is no general way that web browsers can prevent this kind of ham-handed attack. In practice, this is not a common problem on the Web since no one returns to a site that engages in this kind of scripting abuse!

13.9 Other Web-Related JavaScript Embeddings

In addition to client-side JavaScript, the JavaScript language has other web-related embeddings. This book does not cover these other embeddings, but you should know enough about them so that you don't confuse them with client-side JavaScript:

User scripting
: User scripting is an innovation in which user-defined scripts are applied to HTML documents before they are rendered by the browser. Rather than being solely under the control of the page author, web pages can now be controlled by the page visitor as well. The best-known example of user scripting is enabled by the Greasemonkey extension to the Firefox web browser (*http://greasemonkey.mozdev.org*). The programming environment exposed to user scripts is similar

to, but not the same as, the client-side programming environment. This book will not teach you how to write Greasemonkey user scripts, but learning client-side JavaScript programming can be considered a prerequisite to learning user scripting.

SVG

SVG (Scalable Vector Graphics) is an XML-based graphics format that permits embedded JavaScript scripts. Client-side JavaScript can script the HTML document within which it is embedded, and JavaScript code embedded in an SVG file can script the XML elements of that document. The material in Chapters 15 and 17 is relevant to SVG scripting but is not sufficient: the DOM for SVG differs substantially from the HTML DOM.

The SVG specification is at *http://www.w3.org/TR/SVG*. Appendix B of this specification defines the SVG DOM. Chapter 22 uses client-side JavaScript embedded in an HTML document to create an SVG document that is embedded in an HTML document. Since the JavaScript code is outside the SVG document, this is an example of regular client-side JavaScript rather than SVG embedding of JavaScript.

XUL

XUL is an XML-based grammar for describing user interfaces. The GUI of the Firefox web browser is defined with XUL documents. Like SVG, the XUL grammar allows JavaScript scripts. As with SVG, the material in Chapters 15 and 17 is relevant to XUL programming. However, JavaScript code in a XUL document has access to different objects and APIs, and is subject to a different security model than client-side JavaScript code. Learn more about XUL at *http://www.mozilla.org/projects/xul* and *http://www.xulplanet.com*.

ActionScript

ActionScript is a JavaScript-like language (descended from the same ECMAScript specification but evolved in an object-oriented direction) used in Flash movies. Most of the core JavaScript material in Part I of this book is relevant to ActionScript programming. Flash is not XML- or HTML-based, and the APIs exposed by Flash are unrelated to those discussed in this book. This book includes examples of how client-side JavaScript can script Flash movies in Chapters 19, 22, and 23. These examples necessarily include small snippets of ActionScript code, but the focus is on the use of regular client-side JavaScript to interact with that code.

Web Programming/JavaScript

O'REILLY®

JavaScript: The Definitive Guide

This Fifth Edition is completely revised and expanded to cover JavaScript as it is used in today's Web 2.0 applications. This book is both an example-driven programmer's guide and a keep-on-your-desk reference, with new chapters that explain everything you need to know to get the most out of JavaScript, including:

- Scripted HTTP and Ajax
- XML processing
- Client-side graphics using the <canvas> tag
- Namespaces in JavaScript—essential when writing complex programs
- Classes, closures, persistence, Flash, and JavaScript embedded in Java applications

Part I explains the core JavaScript language in detail. If you are new to JavaScript, it will teach you the language. If you are already a JavaScript programmer, Part I will sharpen your skills and deepen your understanding of the language.

Part II explains the scripting environment provided by web browsers, with a focus on DOM scripting with unobtrusive JavaScript. The broad and deep coverage of client-side JavaScript is illustrated with many sophisticated examples that demonstrate how to:

- Generate a table of contents for an HTML document
- Display DHTML animations
- Automate form validation
- Draw dynamic pie charts
- Make HTML elements draggable
- Define keyboard shortcuts for web applications
- Create Ajax-enabled tool tips
- Use XPath and XSLT on XML documents loaded with Ajax
- And much more

Part III is a complete reference for core JavaScript. It documents every class, object, constructor, method, function, property, and constant defined by JavaScript 1.5 and ECMAScript version 3.

Part IV is a reference for client-side JavaScript, covering legacy web browser APIs, the standard Level 2 DOM API, and emerging standards such as the XMLHttpRequest object and the <canvas> tag.

More than 300,000 JavaScript programmers around the world have made this their indispensable reference book for building JavaScript applications.

"A must-have reference for expert JavaScript programmers...well-organized and detailed."
—Brendan Eich, creator of JavaScript

www.oreilly.com

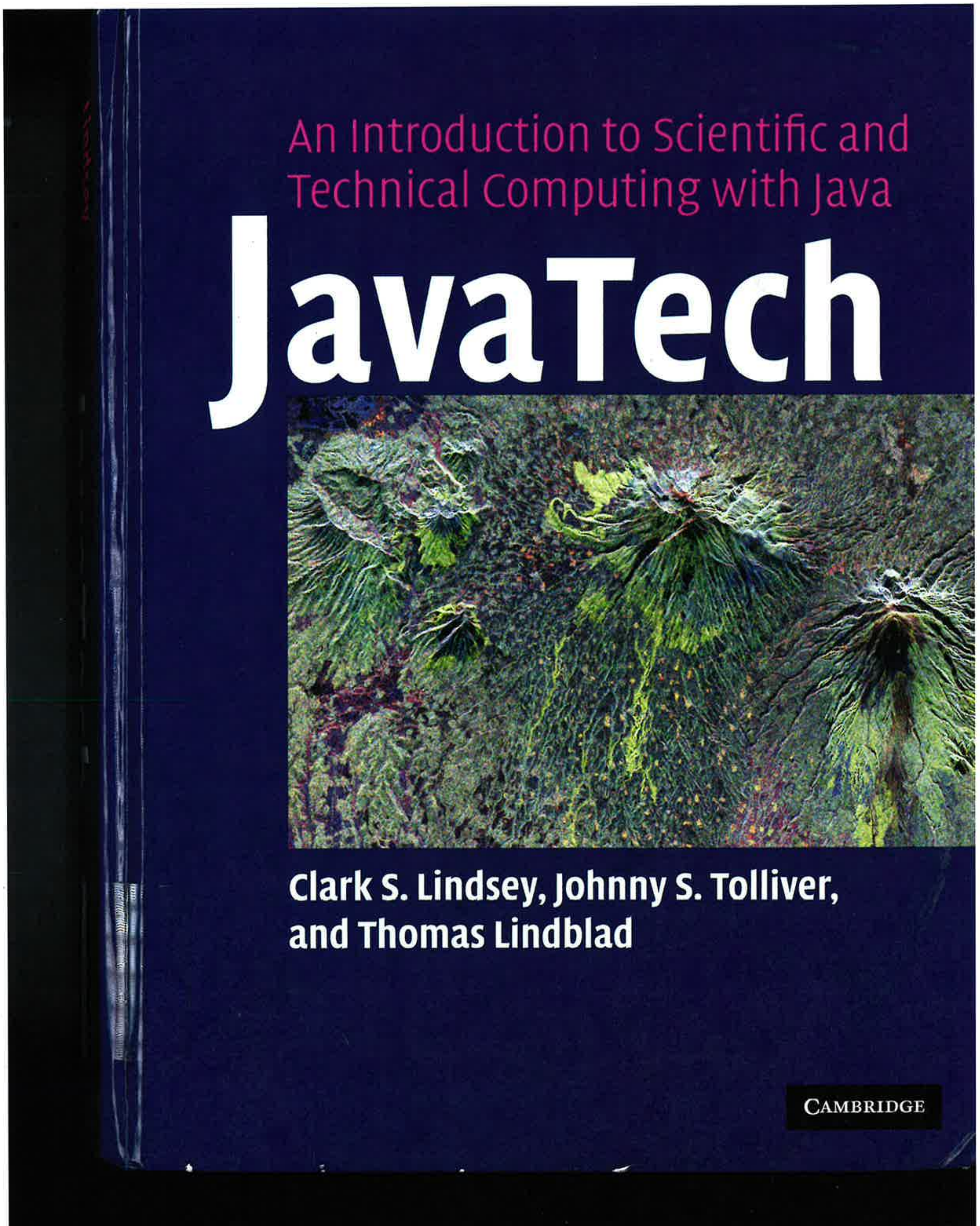
US $49.99 CAN $64.99
ISBN-10: 0-596-10199-6
ISBN-13: 978-0-596-10199-2

54999
9 780596 101992

Safari BOOKS ONLINE ENABLED

Includes FREE 45-Day Online Edition

APPENDIX B



An Introduction to Scientific and Technical Computing with Java

JavaTech

Clark S. Lindsey, Johnny S. Tolliver, and Thomas Lindblad

CAMBRIDGE

JavaTech

An Introduction to Scientific and Technical Computing with Java

Clark S. Lindsey, Johnny S. Tolliver and Thomas Lindblad

CAMBRIDGE
UNIVERSITY PRESS

CAMBRIDGE UNIVERSITY PRESS
Cambridge, New York, Melbourne, Madrid, Cape Town, Singapore, São Paulo

Cambridge University Press
The Edinburgh Building, Cambridge CB2 2RU, UK

Published in the United States of America by Cambridge University Press, New York

www.cambridge.org
Information on this title: www.cambridge.org/9780521821131

© C. S. Lindsey, J. S. Tolliver and T. Lindblad 2005

This publication is in copyright. Subject to statutory exception
and to the provisions of relevant collective licensing agreements,
no reproduction of any part may take place without
the written permission of Cambridge University Press.

First published 2005

Printed in the United Kingdom at the University Press, Cambridge

A catalog record for this publication is available from the British Library

Library of Congress Cataloging in Publication data

ISBN-13 978-0-521-82113-1 hardback
ISBN-10 0-521-82113-4 hardback

Cambridge University Press has no responsibility for the persistence or accuracy of URLs for external or third-party internet websites referred to in this publication, and does not guarantee that any content on such websites is, or will remain, accurate or appropriate.

QA
76.73
.J38
L566
2005

Content

y with the first
g to the bins.
ie histograms,
rth. Functions
ping to deter-
:0 instances of
it is, and you
letails in your

other program,
xtract just that
ulation aspect
vith procedural

onal introduc-
ences between
mory manage-

nd engineering
es and demon-
ial on OOP in
ing techniques

n Introduction

th edn, Sun

ı Guide, March
raining/

Chapter 4
More about objects in Java

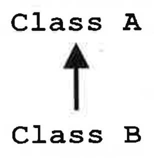
4.1 Introduction

Chapter 3 introduced the basic concepts of classes and objects in Java such as the class definition, instantiation, and object reference. We emphasized the analogy of classes with data types, but the class approach allows for more than just defining a new data type. Java allows you to build upon, or inherit from, a class to create a new child class, or *subclass*, with additional capabilities. In this chapter we introduce *class inheritance* in Java. Inheritance involves the *overriding* (not overloading) of constructors and methods, abstract classes and interfaces, polymorphism, the `Object` class, and the casting of object references to sub- or superclass types. We discuss each of these concepts in detail.

This chapter also includes additional discussion of arrays and how to use them for vectors and matrices in mathematical operations. The chapter ends with a couple of examples of classes for technical applications. We create an improved complex number class and also an enhanced Histogram class.

4.2 Class inheritance

A key feature of object-oriented programming concerns the ability of a class to inherit from an existing class, retaining all the features of the base class but adding new features, thus creating a subclass with increased capabilities. Here class `B` inherits from class `A`, also known as "extending" class `A` (thus the Java keyword `extends`):



```
public class A {
   int i = 0;
   void doSomething () {
      i = 5;
   }
}

class B extends A {
   int j = 0;
   void doSomethingMore () {
      j = 10;
      i += j;
   }
}
```

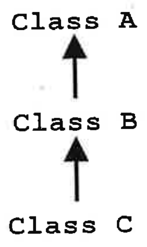
The diagram on the left indicates the class hierarchy. By convention the *superclass* is on top, subclasses are below, and the arrow points upwards from the subclass to the superclass The subclass B has all the data and methods from class A plus the new data and methods added by B. We can think of class B as having the data and methods equivalent to an imaginary class (let's call it "BA") shown here:

```
class BA {
   int i = 0;
   int j = 0;

   void doSomething () {
      i = 5;
   }
   void doSomethingMore () {
      j = 10;
      i += j;
   }
}
```

By using inheritance we get the features of the imaginary class BA without having to duplicate the code from the base class A. We can now create instances of class B and access methods and data in both class B and class A:

```
  . . .
  B b = new B ();          // Create an instance of class B
  b.doSomething ();        // Access a method defined in class A
  b.doSomethingMore ();    // And a method defined in class B
  . . .
```

Another class can, in turn, inherit from class B, as shown here with class C:

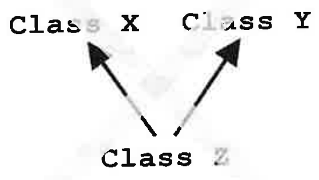


```
class C extends B {
  int k;

  void doEvenMore () {
     doSomething ();
     doSomethingMore ();
     k = i + j;
  }
}
```

Here the `doEvenMore()` method internally calls the `doSomething()` method from class `A` and the `doSomethingMore()` method from class `B`. An instance of class `C` can use the class `C` data and methods and also those of both classes `A` and `B`.

Inheritance does more than just reduce the size of the class definitions. We see shortly that the inheritance mechanism offers several new capabilities including the ability to redefine, or *override*, a method in the superclass with a new one. (The terms *superclass*, *base class*, and *parent class* all mean the same thing and are used interchangeably, as are the terms *subclass* and *child class*.)

Class inheritance in Java is strictly linear. A subclass may extend only one direct superclass, though all of that parent's superclasses get inherited as well in a chaining fashion, as shown in the class `C` example above. Unlike C++, Java does not permit multiple class inheritance, which is inheriting from more than one direct parent class. That is, given two classes `X` and `Y`, it is not possible in Java to create a class `Z` that extends both `X` and `Y`.



There are times that multiple class inheritance could be useful, but it was intentionally omitted by the Java designers because correctly implementing and using multiple class inheritance is fraught with difficulty. Java interfaces, to be discussed later, do permit multiple inheritance, providing many of the benefits of multiple class inheritance without the drawbacks.

4.2.1 Overriding

A common situation is when a class is needed that provides most of the functionality of a potential superclass except one of the superclass methods doesn't do quite the right thing. Adding a new method with a different name in a subclass doesn't really solve the problem because the original superclass method remains accessible to users of the subclass, thereby resulting in a source of errors should a user inadvertently use the original name instead of the new name. What is really needed is a way to change the behavior of that one superclass method without having to rewrite the superclass. Often we may not even have the superclass source code, making rewriting it impossible. Even if we do have the source code, rewriting it would be the wrong approach. That method in the superclass is assumed to be completely appropriate for the superclass and should not be changed. We wish to change the behavior of the method only for instances of our subclass, retaining the existing behavior for instances of the superclass and other subclasses that expect the original behavior of the method.

Java provides just this capability in a technique known as *overriding*. Overriding permits a subclass to provide a new version of a method already defined in a superclass. Instances of the original superclass (and other subclasses) see the original method. Instances of the overriding subclass see the new (overridden) method. In fact, overriding is often the whole reason to create a subclass.

Overriding occurs when a subclass method *exactly* matches the signature (the method name, return type, and parameter types) of a method in a superclass. If the return type is different, a compile-time error occurs. If the parameter list is different, then *overloading* occurs (already discussed in Chapter 3), not overriding. In the next section we discuss the differences, which are very important, but first we give an example of overriding. In the code below, we see that subclass `Child` overrides the method `doSomething()` in class `Parent`:

```
public class Parent {
   int parent_int = 0;
   void doSomething (int i) {
      parent_int = i;
   }
}

class Child extends Parent {
   int child_int = 0;
   void doSomething (int i) {
      child_int = 10;
      parent_int = 2 * i;
   }
}
```

f the func-
ods doesn't
e in a sub-
ass method
ce of errors
ame. What
ass method
the super-
the source
superclass
uld not be
nces of our
s and other

ling. Over-
defined in
es) see the
overridden)
lass.
gnature (the
perclass. If
neter list is
not overrid-
portant, but
at subclass

When we have an instance of class `Child`, an invocation of the method `doSomething()` results in a call to the overridden `doSomething()` code in class `Child` rather than `Parent`:

```
. . .
Parent p = new Parent (); // Create instance of class Parent
Child c = new Child ();    // Create instance of class Child
c.doSomething (5); // The method in class Child is invoked.
p.doSomething (3); // The method in class Parent is invoked.
```

On the other hand, if we call the `doSomething()` method on a `Parent` instance, then the original `doSomething()` code from class `Parent` is invoked. Java automatically invokes the correct method based on the type of the object reference.

The real power of overriding, however, is illustrated by this code:

```
. . .
Parent p = new Child (); // Create an instance of Child
                         // but use a Parent type reference.

p.doSomething (); // Though the Parent type reference
                  // is used, the Child class's doSomething()
                  // is executed.
. . .
```

This code has created an instance of class `Child` but declared it to be of type `Parent`. Doing so is legal when `Child` is a subclass of `Parent`, since `Child` has all the methods and data of type `Parent`. Even though the variable `p` is declared to be the superclass type, it actually references the subclass object. So the subclass method is executed rather than the method in the superclass. This happens because the instance `p` really is of type `Child`, not type `Parent`. The actual type of the object referred to by an object reference is the type that it is "born as," not the type of variable that holds the object reference.

This feature is very useful when, for example, the elements of an array of the base class type contain references to instances of various subclasses. Looping through the array and calling a method that is overridden will result in the method in the subclass being called rather than the method in the base class.

The following code illustrates this so-called *polymorphic* feature of object-oriented languages. We begin with a superclass named `A` and three subclasses `B`, `C`, and `D`, all of which override the `doSomething()` method from `A` (classes `C` and D could be direct subclasses of A or they could be indirect subclasses of A by subclassing B).

```
A[] a = new A[3]; // Class A type array with three elements

a[0] = new B (); // Create an instance of class B but use
                 // an A reference since the array is
                 // type A.
a[1] = new C (); // Ditto for C
a[2] = new D (); // And D

for (int i=0; i < 3; i++) {// Call doSomething() for each
                           // element of the A array.
a[i].doSomething (); // Though the A type reference is used,
                     // the overriding doSomething() method
                     // of the actual referenced object is
                     // invoked.
}
```

It is important to understand that even though the array type is that of the superclass A, the code used for the doSomething() methods is that of the actual object that is referenced in each array element, not the code for the method in the A base class.

4.2.2 Overriding versus overloading

It is important to note how *overriding* differs from *overloading*. The latter refers to reusing the same method name but with a different parameter list and was explained in Chapter 3. Briefly, if a class contains two (or more) methods of the same name but with different parameter lists, all those methods are said to be overloaded. The compiler automatically decides which method to call based on the parameters used when the method is invoked. What was not mentioned in Chapter 3 is that overloading can occur *across* inherited classes. If a subclass reuses a method name from a parent class but changes the parameter list, then the method is still overloaded, just as if both methods appeared in the same class. (Note that via inheritance both methods really do appear in the subclass; the fact that the source code appears in two different places makes no difference.) In over*loading*, the new method does not replace the superclass method; it just reuses the name with a different parameter list. Calling the method with the original parameter list invokes the original method; calling it with the new parameter list invokes the new method.

Confusing overriding and overloading is a vexing error, both for novices and experienced Java developers. If a subclass attempts to override a method in a superclass but doesn't use the exact same parameter list, then the method is really over*loaded*, not over*ridden*. We illustrate this with the following example:

```
public class Parent {
  int i = 0;
  void doSomething (int k) {
      i = k;
  }
}

class Child extends Parent {
   void doSomething (long k) {
       i = 2 * k;
   }
}
```

Here we created class `Child` with the intention of overriding the `doSomething (int k)` method in `class Parent` but we mistakenly changed the `int` parameter to a `long` parameter as shown. Then the `Child` version of `doSomething()` has *overloaded* the `Parent` version, not overridden it. Look what happens when we attempt to call `doSomething()` from an instance of `Child`:

```
. . .
Parent p = new Parent (); // Create a Parnet instance.
Child c = new Child ();   // Create a Child instance.
p.doSomething (5); // The method in Parent is invoked,
                   // as expected.
c.doSomething (3); // The method in Parent, not Child, is
                   // invoked, probably not as expected.
```

The call to `c.doSomething(3)` passes an `int` parameter, not a `long` (a literal 3 is an `int`; to make it a `long`, an `l` or `L` must be appended, as in `3L`). Therefore the overloaded method that takes an `int` is invoked, not the `Child` version expected. Even though we have explicitly asked for `c.doSomething()`, the `int` version of the method named `doSomething()` gets invoked – again, the fact that the source code happens to appear in the superclass makes no difference.

This error is often difficult to uncover. It occurs most often when an overridden superclass method is changed while forgetting to make the same change in the corresponding overriding subclass methods at the same time.

4.2.3 The `@Override` annotation in J2SE 5.0

One of the annotations available with the addition of the metadata facility in Java Version 5.0 (see Chapter 1) greatly reduces the chance of accidentally overloading when you really want to override. The `@Override` annotation tells the compiler that you intend to override a method from a superclass. If you don't get the parameter list quite right so that you're really overloading the method name, the compiler emits a compile-time error. This annotation is used as follows:

```
public class Parent {
   int i = 0;
   void doSomething (int k) {
      i = k;
   }
}

class Child extends Parent {
   @Override
   void doSomething (long k) {
      i = 2 * k;
   }
}
```

The metadata facility in Java 5.0 supports simple and complex annotation types, which are closely related to Java interfaces (discussed in Section 4.5). Some annotation types define member methods and member variables and require parameters when used. However, the `@Override` annotation is just a *marker* interface (see Section 4.5.3). It has no members, and thus accepts no parameters when used, as shown above. It must appear on a line by itself and indicates that the method name on the next line should override a method from a superclass. If the method signature on the next line isn't really an overriding signature, then the compiler complains as follows:

```
Parent.java:10: method does not override a method from its
superclass
   @Override
   ^
1 error
```

By using `@Override` each time you intend to override a method from a superclass, you are safe from accidentally overloading instead of overriding.

4.2.4 Tl

Perhaps yo
However, y
rather than
thing that t
subclass.
When in
superclass
data with t
doSometh
using sup

```
public
   int
   void
      i
   }
}

class C
   int
   void
      j
      /
      s
      j
   }
}
```

You cann
deep as in

```
j = su
```

This usag
overridde
Note th
with the s
useful and
recommen

4.2.4 The `this` and `super` reference operators

Perhaps you need to create a subclass that overrides a method in the base class. However, you want to take advantage of code already in the overridden method rather than rewriting it in the overriding method. That is, you want to do everything that the original method did but add some extra functionality to it for the subclass.

When in a subclass, the special reference variable `super` always refers to the superclass object. Therefore, you can obtain access to overridden methods and data with the `super` reference. In the following code class `Child` overrides the `doSomething()` method in class `Parent` but calls the overridden method by using `super.doSomething()`:

```
public class Parent {
   int i = 0;
   void doSomething () {
      i =5;
   }
}

class Child extends Parent {
   int j=0;
   void doSomething () {
      j = 10;
      // Call the overridden method
      super.doSomething ();
      j += i; // then do something more
   }
}
```

You cannot cascade `super` references to access methods more than one class deep as in

```
  j = super.super.doSomething(); // Error!! Not a valid use of
                                 //super
```

This usage would seem logical but it is not allowed. You can only access the overridden method in the immediate superclass with the `super` reference.

Note that you can also "override" data fields by declaring a field in a subclass with the same name as used for a field in its superclass. This technique is seldom useful and is very likely to be confusing to anyone using your code. Its use is not recommended.

A related concept is known as *shadowing* in which a local variable has the same name as a member variable. For example,

```
public class Shadow {
   int x = 1;
   void someMethod () {
      int x = 2;
      . . .
   }
}
```

Here the x inside someMethod() *shadows* the member variable x in the class definition. The local value 2 is used inside someMethod() while the member variable value 1 is used elsewhere. Such usage is often a mistake, and can certainly lead to hard-to-find bugs. This technique is not recommended. In fact, the variable naming conventions explained in Chapter 5 are designed to prevent accidental shadowing of member variables.

We can also explicitly reference instance variables in the current object with the this reference. The code below illustrates a common technique to distinguish parameter variables from instance or class variables:

```
public class A {
   int x;
   void doSomething (int x) {
      // x holds the value passed in the parameter list.
      // To access the instance variable x we must
      // specify it with 'this'.
      this.x = x;
   }
}
```

Here the local parameter variable shadows the instance variable with the same name. However, the this reference in this.x explicitly indicates that the left-hand side of the equation refers to the instance variable x instead of the local variable x from the parameter list.

4.3 More about constructors

In Chapter 3 we discussed the basics of constructors, including the overloading of constructors. Here we discuss some additional aspects of constructors.

4.3.1 this()

In addition to the this reference, there is also a special method named this() which invokes constructors from within other constructors. When a class holds

overloaded c
basic initiali
tasks. Rather
loaded const
the initializat
For examp

```
class Tes
   int x,
   int i,

   Test (
      x =
      y =
   }

   Test (
      thi
      i =
      k =
   }
}
```

The first con
(the other tv
constructor r
include redu
which execu
The parar
constructor (
types). In thi
int argume
in a construc

4.3.2 su

There is anot
a subclass, it
(we discuss l
tiple overloa
structor gets
with super
For exam
Test1 has

overloaded constructors, typically they include one constructor that carries out basic initialization tasks and then each of the other constructors does optional tasks. Rather than repeating the initialization code in each constructor, an overloaded constructor can invoke `this()` to call another constructor to carry out the initialization tasks.

For example, the following code shows a class with two constructors:

```
class Test {
   int x,y;
   int i,k;

   Test (int a, int b) {
       x = a;
       y = b;
   }

   Test (int a, int b, int c, int d) {
       this (a,b);// Must be in first line
       i = c;
       k = d;
   }
}
```

The first constructor explicitly initializes the values of two of the data variables (the other two variables receive the default 0 value for integers). The second constructor needs to initialize the same two variables plus two more. Rather than include redundant code, the second constructor first invokes `this (a, b)`, which executes the first constructor, and then initializes the other two variables.

The parameter list in the invocation of `this()` must match that of the desired constructor (every constructor must have a unique parameter list in number and types). In this case, `this (a, b)` matches that of the first constructor with two `int` arguments. The invocation of `this()` must be the first executable statement in a constructor and cannot be used in a regular method.

4.3.2 `super()`

There is another special method named `super()`. When we create an instance of a subclass, its constructor plus a constructor in each of its superclasses are invoked (we discuss below the invocation sequence of the constructors). If there are multiple overloaded constructors somewhere in the chain, we might care which constructor gets used. We choose which overloaded superclass constructor we want with `super()`.

For example, in the following code, class `Test2` extends class `Test1`, class `Test1` has a one-argument constructor and a two-argument constructor while the constructor in class `Test2` takes three parameters. Which constructor in the superclass should be invoked? It is unwise to leave it to the compiler to "guess." (Actually, the compiler does not guess; it follows specific rules, which we discuss later.) Let's suppose that our design requires that the two-argument constructor in `Test1` be called. Therefore, the `Test2` constructor invokes the second constructor in class `Test1` by using `super(a, b)`. Had we wanted the one-argument constructor, we would use `super (a)` or `super (b)`.

```
class Test1 {
   int i;
   int j;

   Test1(int i)
   {this.i = i;}

   Test1 (int i, int j) {
      this.i = i;
      this.j = j;
   }
}

class Test2 extends Test1 {
   float x;

   Test2 (int a, int b, float c) {
      super (a, b); // Must be first statement
      x = c;
   }
}
```

As with `this()`, the parameter list identifies which of the overloaded constructors in the superclass to invoke. And as with `this()`, the `super()` invocation must occur as the first statement of the constructor and cannot appear in regular methods.

Do not confuse the `this` and `super` references with the `this()` and `super()` constructor operators. The `this` and `super` references are used to gain access to data and methods in a class and superclass, respectively, while the `this()` and `super()` constructor operators indicate which constructors in the class and superclass to invoke.

4.3.3 Construction sequence

When you instantiate a subclass, the object construction begins with an invocation of the constructor in the topmost base class and *initializes downward* through the constructors in each subclass until it reaches the final subclass constructor. The question then arises: if one or more of the superclasses have multiple constructors, which constructor does the JVM invoke? The answer is that, unless told otherwise with `super()`, the JVM will always choose the zero-argument constructor.

Let's begin with the simplest case of a superclass definition without any constructors. In this case, as we learned in Chapter 3, the compiler automatically generates a zero-argument constructor that does nothing. Almost as simple is the case of a superclass with an explicit zero-argument constructor and no other constructors. In both of these cases, the subclass constructor does not need to explicitly invoke `super()` because the JVM automatically invokes the zero-argument constructor in the superclass – either the zero-argument constructor provided in the superclass source code if there is one, or the default do-nothing "free" constructor if no explicit constructor is provided.

If the superclass contains one or more explicit constructors, then the compiler does *not* generate a free zero-argument constructor. A subclass that does not utilize `super()` to choose one of the existing constructors fails to compile since there is no zero-argument superclass constructor to use. Therefore, the subclass must employ a `super()` with a parameter list matching one of the superclass constructors.

If the subclass also holds several constructors, each must invoke a `super()` to one of the superclass constructors (or perhaps use `this()` to refer to a subclass constructor that does use `super()`). The compiler and JVM figure out the proper sequence of constructors to call as the subclass instance is being built according to which constructor is used with the `new` operator.

The example code here shows two different sequences of constructors invoked for the case of a base class and two subclasses, all with overloaded constructors:

```
public class ConstructApp3 {
   public static void main (String[] args) {

      // Create two instances of Test2
      // using two different constructors.

      System.out.println ("First test2 object");
      Test2 test2 = new Test2 (1.2, 1.3);

      System.out.println ("\nSecond test2 object");
      test2 = new Test2 (true, 1.2, 1.3);
   }
}
```

```
class Test {
   int i;
   double d;
   boolean flag;

   // No-arg constructor
   Test () {
      d = 1.1;
      flag = true;
      System.out.println ("In Test()");
   }

   // One-arg constructor
   Test (int j) {
      this ();
      i = j;
      System.out.println ("In Test(int j)");
   }
}


/** Test1 is a subclass of Test **/
class Test1 extends Test {
   int k;
   // One-arg constructor
   Test1 (boolean b) {
      super (3);
      flag = b;
      System.out.println ("In Test1(boolean b)");
   }
   // Two-arg constructor
   Test1 (boolean b, int j) {
      this (b);
      k = j;
      System.out.println ("In Test1(boolean b, int j)");
   }
}


/** Test2 is a subclass of Test1. **/
class Test2 extends Test1 {
   double x,y;
   // Two-arg constructor
   Test2 (double x, double y) {
     super (false);
```

```
      this
      this
      Syst
   }

   // Thre
   Test2 (
     super
     flag
     Syste
        "I
   }
}
```

The output of

```
First tes
In Test()
In Test(i
In Test1(
In Test2(

Second te
In Test()
In Test(i
In Test1(
In Test1(
In Test2(
```

This illustrat
of the Test2

4.4 Abst

For some app
instantiate. I
the base clas
contain enou
behavior has
base class.

```
        this.x = x;
        this.y = y;
        System.out.println ("In Test2(double x, double y)");
    }

    // Three-arg constructor
    Test2 (boolean b, double x, double y) {
      super (b, 5);
      flag = b;
      System.out.println (
        "In Test2(boolean b, double x, double y)");
    }
  }
```

The output of `ConstructApp3` goes as:

```
First test2 object
In Test()
In Test(int j)
In Test1(boolean b)
In Test2(double x, double y)

Second test2 object
In Test()
In Test(int j)
In Test1(boolean b)
In Test1(boolean b, int j)
In Test2(boolean b, double x, double y)
```

This illustrates the different sequence of constructors invoked according to which of the `Test2` constructors we choose.

4.4 Abstract methods and classes

For some applications we might need a generic base class that we never actually instantiate. Instead, we want always to use subclasses of that base class. That is, the base class handles behavior that is common to all the subclasses but does not contain enough data or behavior to be useful on its own. In a sense, the common behavior has been "factored out" of the subclasses and moved to the common base class.

In the following standard example, we create a base class `Shape`, which provides a method that calculates the area of some 2D shape:

```
public class Shape {
   double getArea () {
      return 0.0;
   }
}
```

The `Shape` class itself does almost nothing. To be useful, there must be subclasses of `Shape` defined for each desired 2D shape, and each subclass should override `getArea()` to perform the proper area calculation for that particular shape. We illustrate with two shapes – a rectangle and a circle.

```
public class Rectangle extends Shape {
   double ht = 0.0;
   double wd = 0.0;

   public double getArea () {
     return ht*wd;
   }
   public void setHeight (double ht) {
     this.ht = ht;
   }
   public void setWidth (double wd) {
     this.wd = wd;
   }
}

public class Circle extends Shape {
   double r =0.0;
   public double getArea () {
     return Math.PI * r * r;
   }
   public void setRadius (double r) {
     this.r = r;
   }
}
```

The subclasses `Rectangle` and `Circle` extend `Shape` and each overrides the `getArea()` method. We could define similar subclasses for other shapes as well. Each shape subclass requires a unique area calculation and returns a `double` value. The default area calculation in the base class does essentially nothing but it must be declared to return a `double` for the benefit of the subclass methods that do return values. Since its signature requires that it return something, it was defined to return 0.0. In practice, since the superclass `Shape` should never be instantiated, only the subclasses, then the superclass `getArea()` will never be called anyway.

The capability to reference instances of `Rectangle` and `Circle` as `Shape` types uses the advantage of polymorphism (see Section 4.2.1) in which a set of different types of shapes can be treated as one common type. For example, in the following code, a `Shape` array passed in the parameter list contains references to different types of subclass instances:

```
void double aMethod (Shape[] shapes) {
   areaSum = 0.0;
   for (int i=0; i < shapes.length; i++) {
       areaSum += shapes[i].getArea ();
   }
}
```

This method calculates the sum of all the areas in the array with a simple loop that calls the `getArea()` method for each instance. The polymorphic feature means that the subclass-overriding version of `getArea()` executes, not that of the base class.

The careful reader will have observed that the technique used above is messy and error-prone. There is no way, for instance, to *require* that subclasses override `getArea()`. And there is no way to ensure that the base class is never instantiated. The above scheme works only if the subclasses and the users of the `Shape` class follow the rules. Suppose someone does instantiate a `Shape` base class and then uses its `getArea()` method to calculate pressure, as in the force per unit area. Since the area is 0.0, the pressure will be infinite (or `NaN`). The Java language can do much better than that.

A much better way to create such a generic base class is to declare a method `abstract`. This makes it explicit that the method is intended to be overridden. In fact, all abstract methods *must* be overridden in some subclass or the compiler will emit errors. No code body is provided for an abstract method. It is just a marker for a method signature, including return type, that must be overridden and given a *concrete* implementation in some subclass.

In the above case, we add the `abstract` modifier to the `getArea()` method declaration in our `Shape` class and remove the spurious code body as shown here:

```
public abstract class Shape {
    abstract double getArea ();
}
```

Note that if any method is declared abstract, the class must be declared abstract as well or the compiler will give an error message. The compiler will not permit an abstract class to be instantiated. An abstract class need not include only abstract methods. It can also include concrete methods as well, in case there is common behavior that should apply to all subclasses. In fact, a class marked `abstract` is not required to include *any* abstract methods. In that case, the `abstract` modifier simply prevents the class from being instantiated on its own. Abstract classes, unlike interfaces (see next section), can also declare instance variables. As an example, our abstract `Shape` class might declare an instance variable `name`:

```
public abstract class Shape {
   String name;
   abstract double getArea ();
   String getName () {
      return name;
   }
}
```

Here each subclass inherits the `name` instance variable. Each subclass also inherits the concrete method `getName()` that returns the value of the `name` instance variable.

When an abstract class does declare an abstract method, then that method must be made concrete in some subclass. For example, let's suppose that class `A` is abstract and defines method `doSomething()`. Then class `B` extends `A` but does not provide a `doSomething()` method:

```
abstract class A {
   abstract void doSomething ();
}
class B extends A {
   // Fails to provide a concrete implementation
   // of doSomething ()
   void doSomethingElse () {. . .}
}
```

In this case, the compiler complains as follows:

```
B is not abstract and does not override abstract method
doSomething() in A class B extends A {
^
```

This message indicates that not overriding `doSomething()` in class `B` is okay if `B` is declared to be abstract too. In fact, that is true. If we don't want `B` to provide `doSomething()`, then we can declare `B` abstract as well:

```
abstrac
   abstı
}
abstrac
   // Dc
   // oi
   void
}
```

This code
instantiatec
subclass oi
methods:

```
class C
   // P:
   void
}
```

4.5 Inte

As discuss
more than

```
class
```

There are
lead to pro
classes inc
parameter

Interfac
problems.
The metho
they are ab
than exte
that impleı
method (oı

In the
run(). Ar
run().

```
abstract class A {
  abstract void doSomething ();
}
abstract class B extends A {
  // Does not provide a concrete implementation
  // of doSomething ()
  void doSomethingElse () {. . .}
}
```

This code compiles without errors. Of course, classes `A` and `B` may never be instantiated directly (since they are abstract). Eventually, there must be some subclass of `A` or `B` that provides a concrete implementation of all the abstract methods:

```
class C extends B {
  // Provides a concrete implementation of doSomething()
  void doSomething () {. . .}
}
```

4.5 Interfaces

As discussed in Section 4.2, Java does not allow a class to inherit directly from more than one class. That is,

```
class Test extends AClass, BClass // Error!!
```

There are situations where multiple inheritance could be useful, but it can also lead to problems; an example is dealing with the ambiguity when the inherited classes include methods and fields with the same identifiers (i.e. the names and parameter lists).

Interfaces provide most of the advantages of multiple inheritance with fewer problems. An interface is basically an abstract class but with *all* methods abstract. The methods in an interface do not need an explicit `abstract` modifier since they are abstract by definition. A concrete class `implements` an interface rather than `extends` it, and a class can implement more than one interface. Any class that implements an interface must provide an implementation of each interface method (or be declared abstract).

In the example below, `Runnable` is an interface with a single method: `run()`. Any class that implements `Runnable` must provide an implementation of `run()`.

```
class Test extends Applet implements Runnable {
  . . .
  public void run () {
    . . .
  }
}
public interface Runnable {
  public void run ();
}
```

To implement multiple interfaces, just separate the interface names with a comma:

```
class Test extends Applet implements Runnable, AnotherInterface
{
  . . .
}
```

If two interfaces each define a method with the same name and parameter list, this presents no ambiguity since both methods are abstract and carry no code body. In a sense, both are overridden by the single method with that signature in the implementing class.

Any class that implements an interface can be referenced as a type of that interface, as illustrated by this code:

```
class User implements Runnable {
  public void run () {
    . . .
  }
}

class Test {
  public static void main (String[] args) {
    Runnable r = new User ();
    . . .
  }
}
```

Here the class `User` implements `Runnable`, so it can be referenced in a variable of type `User` or in a variable of type `Runnable` as shown. The value of using the type `Runnable` instead of `User` is illustrated in the next section.

4.5.1 Ir

The term
because th
is, they ca
For exa
independe
could cor
implemen
which hol

```
public
   pub
}
```

We want t
provide a g
whether a
In the
Relay an
getStat

```
class
  publ
    Sv
    sv
    sv

    fc

    }
  }
}

class
  bool
  // I
  bool
    re
  }
  . .
}
class
  bool
```

4.5.1 Interfacing classes

The term *interface* is a very suitable name for these kinds of abstract classes because they can provide a systematic approach to adding access to a class. That is, they can provide a common *interface*.

For example, say that we have classes `Relay` and `Valve` that are completely independent, perhaps written by two different programmers. The class `Test` could communicate easily with both of these classes if they were modified to implement the same interface. Let's define an interface called `Switchable`, which holds a single method called `getState()`, as in

```
public interface Switchable {
   public boolean getState ();
}
```

We want both the `Relay` and `Valve` classes to implement `Switchable` and provide a `getState()` method that returns a value `true` or `false` that indicates whether a relay or a valve is in the on or off state.

In the code below we show the class `Test` that references instances of `Relay` and `Valve` as `Switchable` types. `Test` can then invoke their respective `getState()` methods to communicate with them.

```
class Test {
  public static void main (String[] args) {
     Switchable[] switches = new Switchable[2];
     switches[0] = new Relay ();
     switches[1] = new Valve ();

     for (int i=0; i < 2; i++) {
       if (switches[i].getState ()) doSomething (i);
     }
  }
}

class Relay implements Switchable {
  boolean setting = false;
  // Implement the interface method getState()
  boolean getState () {
    return setting;
  }

  . . other code . .
}
class Valve implements Switchable {
  boolean valveOpen = false;
```

```
    // Implement the interface method getState()
    boolean getState () {
       return valveOpen;
    }
    . . other code . .
}

interface Switchable {
  boolean getState ();
}
```

So we see that an interface can serve literally to *interface* otherwise incompatible classes together. The modifications required for the classes `Relay` and `Valve` involve only the implementation of the interface `Switchable`. Class `Test` illustrates how we can treat instances of `Relay` and `Valve` both as the type `Switchable` and invoke `getState()` to find the desired information for the particular class. If additional classes that represent other components with on/off states are created for our system simulation, we can ask that they also implement `Switchable`.

Note that if we don't have the source code for `Valve` and `Relay`, we could still create subclasses of them and have those subclasses implement `Switchable`. For example,

```
class SwitchableValve extends Valve implements Switchable {
   boolean getState () {
      . . .
   }
}
```

4.5.2 Interfaces for callbacks

With the C language, programmers often use pointers to functions for tasks such as passing a pointer in an argument list. The receiving function can use the pointer to invoke the passed function. This technique is referred to as a "callback" and is very useful in situations where you want to invoke different functions without needing to know which particular one is being invoked or when library code needs to invoke a function that is supplied by a programmer using the library.

For exampl
an x axis value
example). The
is passed to it
Java, howe
object referen
interfaces for c
face in its para
programmer p
an object refer
required interf
invoked.
In the follo
invokes the ge
of any class th
aFunc(). Th
C. The only di

```
public cla
  public
    Switc
    switc
    switc
    switc

    // Pa
    for (
      aF
    }
  }
  // Rece
  void aF
    if (s
  }
}
. . . See
```

4.5.3 Mor

Interfaces can
ods declared i
Unlike classes

For example, a plotting function could receive a pointer to a function that takes an x axis value as an argument and returns a value for the y axis (*sin(x), cos(x)*, for example). The plotting function could then plot any such function whose pointer is passed to it without knowing explicitly the name of the function.

Java, however, does not provide pointers (actual memory addresses), only object references. A reference cannot refer to a method. Instead, Java provides interfaces for callbacks. In this case, a library method holds a reference to an interface in its parameter list and then invokes a method declared in that interface. The programmer provides a class that implements the required interface and provides an object reference to the library method. When the library method invokes the required interface method, the concrete implementation in the provided object is invoked.

In the following code we see that the `aFunc(Switchable sw)` method invokes the `getState()` method of the `Switchable` interface. An instance of any class that implements the `Switchable` interface can thus be passed to `aFunc()`. This technique provides the same generality as pointer callbacks in C. The only drawback is that a class must implement the interface.

```
public class TestCallBack {
  public static void main(String [] args){
    Switchable[] switches = new Switchable[3];
    switches[0] = new Relay();
    switches[1] = new Relay();
    switches[2] = new Valve();

    // Pass Switchable objects to aFunc ()
    for (int i=0; i < 3; i++) {
      aFunc (switches[i]);
    }
  }
  // Receive Switchable objects and call their getState ()
  void aFunc (Switchable sw) {
    if (sw.getState ()) doSomething ();
  }
}
. . . See previous example for Relay and Valve definitions.
```

4.5.3 More about interfaces

Interfaces can extend other interfaces, much like class inheritance. All the methods declared in the super-interface are effectively present in the sub-interface. Unlike classes, however, interfaces can participate in multiple inheritance. The following code shows an interface extending two interfaces at once using a comma in the `extends` clause:

```
public interface A {. . .}

public interface B {. . .}

public interface C extends A, B {. . .}
```

An interface can also contain data fields, and those fields can be seen by implementing classes. Any data fields in an interface are implicitly `static` and `final` though those qualifiers need not appear. Thus data fields in interfaces are effectively constants and, by convention, are best declared using all uppercase characters.

Placing constants in an interface is a common, though not recommended, practice. As an illustration of the convenience of this technique, consider the `MyConstants` interface shown here:

```
public interface MyConstants {
  final static double G = 9.8;
  final static double C = 2.99792458e10;
}
```

The following `Calculations` class `implements MyConstants` and so can refer to the constants directly:

```
class Calculations implements MyConstants {
  // Can directly use the constants defined
  // in the MyConstants interface
  public double calc (double t) {
    double y = 0.5*G*t*t;
    return y;
  }
}
```

If we instead made `MyConstants` a class, we would need to reference the constants with a class name prefix as follows:

```
double y = 0.5 * MyConstants.G * t * t;
```

This obviously becomes awkward if you have a long equation with lots of constants taken from other classes.

However, despite its usefulness, using an interface just to hold constants is not recommended since it really is an abuse of the interface concept. An interface full

of nothing bu
to do. And a c
anything – it i
would be mo
that the class
For these r
is the recomm
refer to those
the "static im
`import` keyw
Another in
either method
can be useful
operator to de
imply some q
to indicate wh
Section 4.6.3)
We discuss
that interface
any class can
implementatio
must also be p

4.6 More

In this section
objects with ar
method.

4.6.1 Conv

In Chapter 2 we
operation and t
same concepts
as with primiti
Consider a sup

```
class Fru
class Pine
```

Let `f` be a vari
assign the `Pin`

of nothing but constants does not define a type, as a proper interface is expected to do. And a class that "implements" such an interface isn't really *implementing* anything – it is just *using* the constants in the interface (perhaps a `uses` keyword would be more appropriate). Seeing the `implements` keyword should imply that the class actually implements something.

For these reasons, the use of a class instead of an interface to define constants is the recommended practice, accepting the need for the more verbose syntax to refer to those constants. We note that J2SE 5.0, in fact, solves this problem with the "static import" facility, which we explain in Chapter 5 after discussing the `import` keyword.

Another interesting feature of interfaces is that an interface need not contain either method declarations or data. It can be completely empty. Such an interface can be useful as a "marker" of classes. That is, you can use the `instanceof` operator to determine if a class is of the particular marker type, which then can imply some quality of the class. One can use the empty `Cloneable` interface to indicate whether a class overrides the `clone()` method from `Object` (see Section 4.6.3) to make copies of instances of the class.

We discuss access rules and modifiers in the next chapter but here we note that interface methods and constants are implicitly `public`. This means that any class can access the methods and constants in the interface. The concrete implementations of interface methods in classes that implement the interface must also be `public` otherwise the compiler will complain.

4.6 More about classes

In this section we continue our introduction to the basics of class definitions and objects with an examination of casting, the `Object` class, and the `toString()` method.

4.6.1 Converting and casting object references

In Chapter 2 we discussed the topic of mixing different primitive types in the same operation and the need in some cases to explicitly cast one type into another. The same concepts apply when dealing with objects instead of primitives. Sometimes, as with primitives, the type conversion is automatically handled by the compiler. Consider a superclass `Fruit` with a subclass `Pineapple`:

```
class Fruit {. . .}
class Pineapple extends Fruit {. . .}
```

Let `f` be a variable of type `Fruit` and `p` be of type `Pineapple`. Then we can assign the `Pineapple` reference to the `Fruit` variable:

```
class Conversion {
   Fruit f;
   Pineapple p;
   public void convert () {
      p = new Pineapple ();
      f = p;
   }
}
```

The compiler automatically handles the assignment since the types are compatible. That is, the type `Fruit` can "hold" the type `Pineapple` since a `Pineapple` "is a" `Fruit`. Such automatic cases are called *conversions*.

A related automatic conversion is with interfaces. Let the class `Fruit` implement the `Sweet` interface:

```
interface Sweet {. . .}
class Fruit implements Sweet {. . .}
```

Then we see that a variable of type `Fruit` can be automatically converted to a variable of type `Sweet`. This makes perfect sense since a `Fruit` "is" `Sweet`.

```
Fruit f;
Sweet s;
public void good_convert () {
  s = f; // legal conversion from class type to interface type
}
```

However, an attempt to convert from the interface type to the class type does not compile:

```
public void bad_convert () {
  f = s; // illegal conversion from interface type to class type
}
```

As with primitives, if the compiler cannot perform an automatic conversion, an explicit cast is required. In most cases you can force the compiler to permit the desired type conversion by using a cast. Like with primitive types, the class type that an object is being cast to is enclosed in parentheses in front of the object reference. Doing so essentially tells the compiler to ignore the apparent type incompatibility and proceed anyway. If the types really are incompatible then runtime errors will ensue.

For example, let `BClass` be a subclass of `AClass`. Let `AClass` hold `aMethod()`, which, of course, is inherited by `BClass`. In addition, `bMethod()` is a new method in `BClass`.

```
class A
  void
}

class B
   voi
}
```

In the follo
as a parame
of BClass,

```
public v
  obj.a
  if (o
}
```

We see that
whether that
However, to
use the ins
then cast to
it cannot fin

4.6.2 Ca

The casting
There are co
to catch atten
we have clas
the other an
nonsensical t
permit it eve
types" error
Casts that
class or to on
be cast to aln
type. There a
details), but t
The compi
time rules pe
These runtim
with the new
at runtime.

```
class AClass {
  void aMethod () {. . .}
}

class BClass extends AClass {
   void bMethod () {. . .}
}
```

In the following code, `miscMethod()` is declared to receive an `AClass` object as a parameter. When used, the actual object passed in might in fact be an instance of `BClass`, which is perfectly legal since `BClass` is a subclasses of `AClass`.

```
public void miscMethod (AClass obj) {
  obj.aMethod ();
  if (obj instanceof BClass) ((BClass)obj).bMethod ();
}
```

We see that we can invoke `aMethod()` on the object received in the parameter list whether that object is an `AClass` or a `BClass` since both types have this method. However, to invoke the `bMethod()`, we need first to check the object type. We use the `instanceof` operator to find out if the object really is a `BClass` and then cast to `BClass` if appropriate. Without the cast the compiler complains that it cannot find `bMethod()` in the `AClass` definition.

4.6.2 Casting rules

The casting rules can be confusing, but in most cases common sense applies. There are compile-time rules and runtime rules. The compile-time rules are there to catch attempted casts in cases that are simply not possible. For instance, suppose we have classes `A` and `B` that are completely unrelated – i.e. neither inherits from the other and neither implements the same interface as the other, if any. It is nonsensical to attempt to cast a `B` object to an `A` object, and the compiler does not permit it even with an explicit cast. Instead, the compiler issues an "inconvertible types" error message.

Casts that are permitted at compile-time include casting any object to its own class or to one of its sub- or superclass types or interfaces. Almost anything can be cast to almost any interface, and an interface can be cast to almost any class type. There are some obscure cases (see the Java Language Specification for the details), but these common sense rules cover most situations.

The compile-time rules cannot catch every invalid cast attempt. If the compile-time rules permit a cast, then additional, more stringent rules apply at runtime. These runtime rules basically require that the object being cast is compatible with the new type it is being cast to. Else, a `ClassCastException` is thrown at runtime.

4.6.3 The `Object` class

All classes in Java implicitly extend the class `Object`. That is,

```
public class Test
{. . .}
```

is equivalent to

```
public class Test extends Object
{. . .}
```

So, all Java objects are instances of `Object`. This ability to treat all objects as one type provides the ultimate in polymorphism. An example of this usage is the `ArrayList` class, which is a part of the `java.util` package (we discuss Java packages in Chapter 5). The `ArrayList` class can hold any object type. The `ArrayList.add()` method is used to input objects into the `ArrayList`. The parameter list for the `add()` method is declared to receive an `Object` parameter. That way, any object type can be added, since all object types always inherit from the `Object` base class. When an element is retrieved from the `ArrayList`, it is of type `Object` and should be cast to the type needed.

A simpler example is the following case, where the parameter type of `miscMethod()` is `Object` so any class whatsoever can be provided in a method call to `miscMethod()`. Inside `miscMethod()` we decide what type the received object reference really is and call appropriate methods based on that type. Except for the case where we want to invoke a method belonging to the `Object` class, we need to cast the object to one of the classes that we expect as a parameter before we can invoke a method or access a field in that class.

```
public void miscMethod (Object obj) {
   if (obj instanceof AClass) ((AClass)obj).aMethod ();
   if (obj instanceof BClass) ((BClass)obj).bMethod ();
   if (obj instanceof CClass) ((CClass)obj).cMethod ();
}
```

The `Object` class provides several methods that are useful to all of its subclasses. A subclass can also override these methods to provide behavior unique to the particular subclass. These methods include:

- `clone ()` – produces copies of an object. (See Web Course Supplements.)
- `equals (Object obj)` – tests whether an object is equal to the object `obj`. The default is to test simply whether `obj` references the same object (i.e. a shallow equals), not whether two independent objects contain identical properties. This method is often overridden to perform a deep equals as in the `String` class, which tests whether the strings actually match.

objects as
sage is the
scuss Java
type. The
ist. The
parameter.
herit from
yList, it

r type of
a method
e received
pe. Except
ect class,
parameter

();
();
();

subclasses.
ique to the

)
t obj. The
low equals),
hod is often
whether the

- `toString ()` – provides a string representation of this object. The default for a class consists of a string constructed from the name of the class plus the character `"@"` plus a hash code of the object in hex format. This method is often overridden to provide more illuminating information. (See the next section.)
- `finalize ()` – called by the garbage collector when there are no more references to this object. You can override this method to take care of any housecleaning operations needed before the object disappears.
- `getClass ()` – gets the runtime class of the object, returned as a `Class` type (see the Web Course Chapter 5: *Supplements* section for a discussion of the `Class` class).
- `hashCode ()` – generates a hash code value unique for this object.

The following methods involve thread synchronization that we introduce in Chapter 8. They can only be called from within a synchronized method or code block:

- `notify ()` – called by a thread that owns an object's lock to tell a waiting thread, as chosen by the JVM, that the lock is now available.
- `notifyAll ()` – similar to `notify()` but wakes all waiting threads and then they compete for the lock.
- `wait ()` – the thread that owns the lock on this object releases the lock and then waits for a `notify()` or `notifyAll()` to get the lock back.
- `wait (long msecs)` – same as `wait()` but if a notify fails to arrive within the specified time, it wakes up and starts competing for the lock on this object anyway.
- `wait (long msecs, int nanosecs)` – same as `wait (long msecs)` but specified to the nanosecond.

(We note that most operating systems do not provide a clock that is accurate to a nanosecond and some not even to a few milliseconds.)

4.6.4 Objects to strings

We discussed in Chapter 3 how to convert primitive types to and from strings. You can also convert any Java object to a string. If you just print any object, as in

```
System.out.println (someObjectReference);
```

then that object's `toString()` method is called automatically to produce string output. All objects inherit the `toString()` method from the `Object` class. This default version of `toString()` from `Object` produces a string beginning with the class name with certain data values appended to it.

However, the `toString()` method typically is overridden by most classes to provide output in a more readable format customized for that class. Most of the classes in the Java core class libraries provide sensible `toString()` methods, and classes that you write should too for convenience when printing.

You can call the `toString()` method directly, or, alternatively, the "+" operator calls the `toString()` method whenever the variable refers to an object. For example, consider

```
Double aDouble = 5.0;
String aDoubleString = "aDouble = " + aDouble;
```

The plus operator in the second line invokes the `toString()` method of the `Double` object `aDouble`. This results in `aDoubleString` referencing the string "`aDouble = 5.0`".

4.7 More about arrays

Here we look at other aspects of Java arrays and at tools to use with them. Note that like much of Java syntax, arrays at first glance seem very similar to those in C/C++. However, there are several differences from these languages in how Java arrays are built and how they work.

4.7.1 Object arrays

In the previous chapter we introduced arrays of primitive types, which generally behave in the manner that is expected of such arrays. For example, to create an array of ten integers we could use the following:

```
int[] iArray = new int[10];
```

This sets aside ten `int` type memory locations, each containing the value 0.

For arrays of objects, however, the array declaration only creates an array of references for that particular object type. It does not create the actual objects of that type. Creating the objects themselves requires an additional step. For example, let's say we want to create an array of five `String` objects. We first create a `String` type array:

```
String[] strArray = new String[5];
```

When the array is created, five memory locations are set aside to contain object references of the `String` type with the expectation that each reference will eventually "point" to a `String` object. But initially, each element contains the special `null` reference value; that is, it points *nowhere*. So if we followed the above declaration with an attempt to use a `String` method, as in

```
int numChars = strArray[0].length ();
```

an error message results:

```
Exception in thread "main" java.lang.NullPointerException at
ArrayTest.main (ArrayTest.java:8)
```

Before using
element to re

```
strArray
strArray
strArray
strArray
strArray
```

This code se
Note that
objects:

```
strArray
```

That is, the s

4.7.2 Arr

A copy of an
as shown her

```
System.a
```

Here `src` is
same type).
`src_posit`
ber of eleme
situation occ
beyond their
thrown at ru
object arrays
source array.

4.7.3 Mu

In Java, mult
reference to
array as follo

```
String[
```

This is equiv

```
String
str[0] =
str[1] =
str[2] =
```

ly, the "+"
to an object.

:thod of the
rencing the

them. Note
.lar to those
ages in how

:h generally
to create an

value 0.
an array of
tual objects
al step. For
:ts. We first

ntain object
'erence will
:ontains the
'ollowed the

:eption at

Before using the array elements, we must first create an object for each array element to reference. For example,

```
strArray[0] = new String ("Alice");
strArray[1] = new String ("Bob");
strArray[2] = new String ("Cindy");
strArray[3] = new String ("Dan");
strArray[4] = new String ("Ed");
```

This code sets each element to reference a particular string.

Note that there is an alternative declaration that only works for `String` objects:

```
strArray[0] = "Alice";
```

That is, the string literal `"Alice"` is equivalent to `new String ("Alice")`.

4.7.2 Array copying

A copy of an array can be made with the static method `System.arrayCopy()` as shown here:

```
System.arraycopy (Object src, int src_position,
                  Object dst, int dst_position, int length)
```

Here `src` is the array to be copied and `dst` is the destination array (of the same type). The copy begins from the array element at the index value of `src_position` and starts in destination at `dst_position` for `length` number of elements. If the value of the `length` parameter is too long, or if any situation occurs such that either the source or destination arrays are accessed beyond their actual array length, then an `IndexOutOfBoundsException` is thrown at runtime. This optimized method works for primitive arrays as well as object arrays. It even handles the case where the destination array overlaps the source array.

4.7.3 Multi-dimensional arrays

In Java, multi-dimensional arrays are arrays of arrays. That is, each element is a reference to an array object. For example, we could declare a two-dimensional array as follows:

```
String[][] str = new String[3][2];
```

This is equivalent to

```
String [][] str = new String[3][];
str[0] = new String[2];
str[1] = new String[2];
str[2] = new String[2];
```

However, we don't need to keep the sub-array lengths the same. This also works:

```
str[0] = new String[2];
str[1] = new String[33];
str[2] = new String[444];
```

We can combine the string array declaration and initialization, as in

```
str[0] = new String[]{"alice", "bob"};
str[1] = new String[]{"cathy", "don", "ed"};
str[2] = new String[]{"fay", "grant", "hedwig", "ward"};

System.out.println ("str[1][2],str[2][3] = " +
str[1][2] + str[2][3]);
```

The print statement would show

```
str[1][1],str[2][3] = edward
```

4.7.4 More about arrays as objects

As mentioned earlier, arrays in Java are objects. An array inherits `Object` and possesses an accessible property – `length` – that gives the number of elements in the array. For example, if a method uses `Object` as a parameter, as in

```
void aMethod (Object obj) {. . .}
```

then an array can be passed as the actual parameter since an array is a subclass of `Object`:

```
. . .
int[] i_array = new int[10];
aMethod (i_array);
. . .
```

To make arrays appear in a convenient and familiar form (as in C, for example), the language designers provided brackets as the means of accessing the array elements as already seen above. Without brackets, an array class would have to provide a method such as `getElementAtIndex()` to access array elements. For example,

```
String string_one = str_array.getElementAtIndex (1);
```

Fortunately, the simpler syntax using brackets was chosen instead:

```
String string_one = strArray[1];
```

Since arrays are objects, arrays are somewhat more complicated in Java than in other languages, but the class structure also provides important benefits. For example, each use of an array element results in a check on the element number,

and if the e
run-time ex
Thus, un
and write to
program bu
since the pr
hand, there
up when do

4.7.5 Ma

Vector and r
engineering
out operatio
Note tha
`java.uti`
discussed al
both adding
useful, but t

4.7.5.1 M

The elemen
vector, as in

```
double[
double[
```

We then ne
product:

```
double
  doubl
  for (
    do
  }
  retur
}
```

Note that a
null and tha
Several n
carry out ve
of these.

and if the element exceeds the declared length of the array, an out of bounds run-time exception is thrown.

Thus, unlike in C or C++, a program cannot run off the end of an array and write to places in memory where it should not. This avoids a very common program bug and source of security attacks that can be difficult to track down since the problem may not show up until well after the write occurs. On the other hand, there is some performance penalty in the bounds checking that can show up when doing intensive processing with arrays.

4.7.5 Mathematical vectors and matrices

Vector and matrix operations are obviously standard tools throughout science and engineering. Here we look at some ways to use Java arrays to represent and carry out operations for vectors and matrices.

Note that the Java core language includes a class called `Vector` in the `java.util` package (see Chapter 10). `Vector` is similar to the `ArrayList` discussed above (see Section 4.6.3); both provide a dynamic list that allows for both adding and removing elements. `ArrayList` and `Vector` are often quite useful, but they are slow and not intended for mathematical operations.

4.7.5.1 Mathematical vectors

The elements of a floating-point array can represent the component values of a vector, as in

```
double[] vec1 = {0.5,0.5,0.5};
double[] vec2 = {1.0,0.0,0.2};
```

We then need methods to carry out various vector operations such as the dot product:

```
double dot (double[] a, double[] b) {
  double dot_prod = 0.0;
  for (int i=0; i < a.length; i++) {
    dot_prod += a[i]*b[i];
  }
  return dot_prod;
}
```

Note that a more robust method would check that the vector arguments are not null and that the array lengths are equal.

Several numerical libraries are available that provide classes with methods to carry out vector operations. The Web Course Chapter 4 provides links to several of these.

4.7.5.2 Matrices

The obvious approach for matrices is to use arrays with two indices:

```
double[][] dMatrix = new double[n][m];
```

However, as indicated by the discussion in Section 4.7.2, this does not produce a true two dimensional array in memory but is actually a one-dimensional array of references to other one-dimensional arrays, each of which can be located in a different area of memory.

In the C language, moving from one element to the next in a 2D array requires only incrementing a memory pointer. This does not apply for Java, which uses an indirect referencing approach that causes a performance penalty, especially if the matrix is used in intensive calculations.

One approach to ameliorate this problem to some extent is to use a 1D array. The code below shows how one might develop a matrix class to use a 1D array for 2D operations. A sophisticated compiler can optimize such a class and in some cases provide better performance than a standard Java two-dimensional array.

```
public class Matrix2D {
  private final double[] fMat;

  private final int fCols;
  private final int fRows;
  private final int fCol;
  private final int fRow;

  public Matrix2D (int rows, int cols) {
    fCols = cols;
    fRows = rows;
    fMat= new double[rows * cols];
  }

  /** r = row number, c = column number **/
  public double get (int r, int c) {
    return fMat[r * fCols + c];
  }

  /** r = row number, c = column number **/
  public double set (int r, int c, double val) {
    fMat[r * fCols + c] = val;
  }

  . . . other methods, e.g. to fill the array, access a
  subset of elements, etc.
}
```

4.8 Impr

In the Chapt
complex nui
like to add t
rather than n
can still use
number clas

```
public c

  double
  double
  /** Cc
   **/
  Comple
    real
  }

  /** Ge
  public
  {retur
  public
  {retur

  /** De
  public
    real
    imag
  }

  /** De
  public
    real
    imag
  }

  /** De
   *  a
   **/
  public

    doub
    doub
    retu
  }
```

4.8 Improved complex number class

In the Chapter 3 we created a class with the bare essentials needed to represent complex numbers. Here we expand on that class. For example, we would often like to add two complex numbers and put the sum into another complex number rather than modify one of the current complex objects. Because of overloading we can still use the `add()` method name. A new, improved version of our complex number class appears here:

```
public class Complex {

  double real;
  double imag;
  /** Constructor that initializes the real & imag values
   **/
  Complex (double r, double i) {
    real = r; imag = i;
  }

  /** Getter methods for real & imaginary parts **/
  public double getReal ()
  {return real;}
  public double getImag ()
  {return imag;}

  /** Define an add method **/
  public void add (Complex cvalue) {
    real = real + cvalue.real;
    imag = imag + cvalue.imag;
  }

  /** Define a subtract method. **/
  public void subtract (Complex cvalue) {
    real = real - cvalue.real;
    imag = imag - cvalue.imag;
  }

  /** Define a static add method that returns a
   *  a new Complex object with the sum.
   **/
  public static Complex add (Complex cvalue1,
                             Complex cvalue2) {
    double r = cvalue1.real + cvalue2.real;
    double i = cvalue1.imag + cvalue2.imag;
    return new Complex (r, i);
  }
  /** Define a static subtract method that returns a
   * a new Complex object with the difference.
   **/
  public static Complex subtract (Complex cvalue1,
                                  Complex cvalue2) {
    double r = cvalue1.real - cvalue2.real;
    double i = cvalue1.imag - cvalue2.imag;
    return new Complex (r, i);
  }
} // class Complex
```

Here the new static `add()` and `subtract()` methods each create a new complex object to hold the sum and difference, respectively, of the two input complex objects. The operand objects are unchanged by the method.

As we discussed in Chapter 3, a static method is invoked by giving the name of the class and the dot operator. Unfortunately, in Java, unlike C++, we cannot override the + operator and create a special + operator for complex addition. The following code shows how to add two complex numbers together using our static `add()` method:

```
public class ComplexTest {
public static void main (String[] args) {
   // Create complex objects
   Complex a = new Complex (1.0, 2.1);
   Complex b = new Complex (3.3, 1.2);

   Complex c = Complex.add (a, b); // c now holds a + b

    . . . other code . . .
   }
}
```

The Web Course Chapter 4 gives a more complete version of the class (e.g. it includes modulus, multiplication, etc.).

4.9 Random number generation

Random values can be obtained from the `Math` class using the method

```
public static double random ()
```

This method produces pseudo-random double values in the range

```
0.0 <= r < 1.0
```

The first t
current tin
The ja
generators
the option:
The me

- nextInt
- nextInt
- nextBoc
- nextGau

The last th

4.9.1 R

The Rando
the constru
the algoritl
Randon
that eventu
sequence.
eventually
to the rand
are said to
To insur
all implem
returns the
The line

x_{i+1} =

As discuss
produce ra
long. The l
Also, if
as points in
but instead
and possib
shuffle the
In Java t

a = 0x
c = 11
m = 2^48

The actual

The first time it is invoked, it initializes the seed with a value derived from the current time.

The `java.util.Random` class provides a more extensive set of random generators. Two constructors – `Random()` and `Random (long seed)` – offer the options of initialization from the current time or from a specific `seed` value.

The methods in the `Random` class include:

- `nextInt ()` – integers in the range `0 <= r < 2**32`
- `nextInt (int n)` – integers in the range `0 <= r < n`
- `nextBoolean (int n)` – randomly chosen true/false
- `nextGaussian ()` – random double values with mean 0.0 and sigma of 1.0

The last three methods first became available with Java 1.2.

4.9.1 Random number algorithm

The `Random` class uses a *linear congruential algorithm* [1,2] with a 48-bit seed. If the constructor `Random (long)` or the `setSeed (long)` method is invoked, the algorithm uses only the lower 48 bits of the seed value.

Random number generator formulas actually produce a sequence of numbers that eventually repeat. For the same seed value a formula always produces the same sequence. A seed simply selects where in the sequence to start. The generator will eventually repeat that seed value and start the same sequence again. Compared to the randomness of physical fluctuations, such as in radio noise, these formulas are said to produce *pseudo-random* numbers.

To insure that applications ported to different platforms give the same results, all implementations of Java must use the same algorithm so that the same seed returns the same sequence regardless of the platform.

The linear congruential formula in Java goes as

$$x_{i+1} = (a * x_i + c) \bmod m$$

As discussed in the references, you should use such formulas with care. They can produce random number sequences of a length up to `m` but not necessarily that long. The length depends on the set of `a`, `c`, and `m` values chosen.

Also, if you grab consecutive sequences of numbers of `K` length, and plot them as points in `K`-dimensional space, they do not fully populate the volume randomly but instead lie on `K-1`dimensional planes. There are no more than $m^{1/K}$ planes and possibly less. If you need to create points in a space this way, you should shuffle the values obtained from the generator. [2]

In Java the values in the linear congruential formula in `Random` are

$$a = 0x5DEECE66DL$$
$$c = 11$$
$$m = 2^{48} - 1.$$

The actual code in `next (int bits)` goes as

```
  synchronized protected int next (int bits) {
    seed = (seed * 0x5DEECE66DL + 0xBL) & ((1L << 48) - 1);
    return (int)(seed >>> (48 - bits));
  }
```

Here the `mod` operation comes via the `AND` operation since `m` in this case has all 47 bits set to 1.

This method is `protected` (see Section 5.3.3, Access Rules). The public random number methods accessible by all classes use the `next()` method. For example, `nextInt()` simply includes the statement

```
  return next (32);
```

The `nextLong()` method invokes `next(32)`, shifts the result by 32 bits to the left, invokes `next(32)` again and then `OR`s the two values together to obtain a 64-bit random number:

```
  return ((long)next (32) << 32) + next (32);
```

The `nextFloat()` method provides values in the range `0.0f <= x < 1.0f`:

```
  return next (24) / ((float)(1 << 24));
```

The `nextDouble()` method provides values in the range `0.0d <= x < 1.0d` using the statement

```
return (((long)next (26) << 27) + next (27))/(double)(1L << 53)
```

The `nextBoolean()` method uses the statement

```
  return next (1)!= 0;
```

See the `java.util.Random` class specification for more detailed descriptions of the algorithms used for these and the other `next`*`Xxx`*`()` methods.

4.10 Improved histogram class

Here we make a subclass of the `BasicHist` class discussed in Chapter 3. The class definition below shows that `BetterHist` inherits from `BasicHist`, obtaining the properties of the latter while providing new capabilities.

Note how the constructor invokes `super()` to select a constructor in the base class. Also, we see how the new methods in the subclass can access the data variables in the base class. (In the next chapter we discuss access modifiers such as `private`, which prevents subclasses from accessing a field or method.)

We add several methods to our histogram that provide various parameters specifying the histogram. Also, a calculation of the mean and standard deviation of the distribution in the histogram is included.

- 1);

:ase has all

The public
ıethod. For

bits to the
to obtain a

< 1.0f:

: < 1.0d

L << 53)

escriptions

Chapter 3.
sicHist,

in the base
ss the data
lifiers such
thod.)
parameters
d deviation

```
/** A simple histogram class to count the frequency of
  * values of a parameter of interest. **/
class BetterHist extends BasicHist
{
   /** This constructor initializes the basic elements of
    *  the histogram.
    **/
   public BetterHist (int numBins, double lo, double hi) {
     super (numBins, lo, hi);
   }

   /** Get the low end of the range. **/
   public double getLo ()
   {return lo;}

   /** Get the high end of the range. **/
   public double getHi ()
   {return hi;}

   /** Get the number of entries in the largest bin. **/
   public int getMax () {
     int max = 0;

     for (int i=0; i < numBins; i++)
       if (max < bins[i]) max = bins[i];
     return max;
   }
   /** Get the number of entries in the smallest bin. **/
   public int getMin () {
     int min = getMax ();

     for (int i=0; i < numBins; i++)
       if (min > bins[i]) min = bins[i];
     return min;
   }

   /** Get the total number of entries **/
   public int getTotal () {
     int total = 0;
     for (int i=0; i < numBins; i++)
       total += bins[i];
     return total;
   }
   /** Get the average and std. dev. of the distribution. **/
   public double [] getStats () {
     int total = 0;
```

```
      double wtTotal = 0;
      double wtTotal2 = 0;
      double [] stat = new double[2];
      double binWidth = range/numBins;

      for (int i=0; i < numBins; i++) {
         total += bins[i];
         double binMid = (i - 0.5) * binWidth + lo;
         wtTotal += bins[i] * binMid;
         wtTotal2 += bins[i] * binMid * binMid;
      }

      if (total > 0) {
        stat[0] = wtTotal/total;
        double av2 = wtTotal2/total;
        stat[1] = Math.sqrt (av2 - stat[0]*stat[0]);
      }
      else {
        stat[0] = 0.0;
        stat[1] = -1.0;
      }
      return stat;
   } // getStats
} // class BetterHist
```

4.11 Understanding OOP

Chapters 3 and 4 present the fundamentals of class definitions and objects. In Chapter 5 we look at how classes are organized into files and directories and how the JVM locates classes. If you find that object-oriented programming (OOP) remains somewhat vague, your understanding of the concepts involved will deepen as you see OOP techniques applied to graphics, threading, I/O, and other areas in subsequent chapters. We return to class structure, design, and analysis in Chapter 16 where we give a brief overview of the Unified Modeling Language (UML). UML provides a systematic approach to the design of classes and to analysis of the interactions among objects. We then use UML to design a set of classes for a distributed computing example.

4.12 Web Course materials

The Web Course Chapter 4: *Supplements* section provides more discussion of inheritance and the overriding and overloading features. There is also discussion of security aspects of Java including the checking of code by the JVM during class loading. It also gives a brief overview of the security manager.

The Cha
programs f
number ger
section con

Referenc

[1] Donald
Volume
[2] W. H. Pr
C: The A
available

Resource

Ronald Mak,
Computing

The Chapter 4: *Tech* section provides additional discussion and demonstration programs for the vector/matrices in Java, the complex number class, random number generation, and the improved histogram class. The Chapter 4: *Physics* section continues with a tutorial on numerical computing techniques with Java.

References

[1] Donald Knuth, *The Art of Computer Programming: Semi-numerical Algorithms Volume 2*, 3rd edn, Addison-Wesley, 1997.

[2] W. H. Press, B. P. Flannery, S. A. Teukolsky and W. T. Vetterling, *Numerical Recipes in C: The Art of Scientific Computing*, Cambridge, 1992. (Subsequent versions are available for Fortran 90 and C++.)

Resources

Ronald Mak, *Java Number Cruncher: The Java Programmer's Guide to Numerical Computing*, Prentice Hall, 2003.

bjects. In
and how
ıg (OOP)
lved will
and other
l analysis
anguage
es and to
n a set of

ussion of
iscussion
M during

Chapter 8
Threads

8.1 Introduction

Threads in Java are processes that run in parallel within the Java Virtual Machine. When the JVM runs on a single real processor the parallelism is, of course, only apparent because of the high speed switching of threads in and out of the processor. Yet even in that case, threading can provide significant advantages. For example, while one thread deals with a relatively slow I/O operation such as downloading a file over the network, other threads can do useful work. Threads can assist in modularizing program design. An animation can assign different threads to rendering the graphics, to sound effects, and to user interactions so that all of these operations appear to take place simultaneously. Furthermore, a JVM can assign Java threads to native OS threads (but isn't required to) and on a multiprocessor system it could thus provide true parallel performance.

Java makes the creation and running of threads quite easy. We will concentrate on the basics of threading and only briefly touch on the subtle complications that arise when multiple threads interact with each other and need to access and modify common resources. Such situations can result in *data race*, *deadlock*, and other interference problems that result in distorted data or hung programs.

8.2 Introduction to threads

In Java you can create one or more threads within your program just as you can run one or more programs in an operating system [1–4]. Most JVMs, in fact, take great advantage of threads for such tasks as input/output operations and user-interface event handling. Since the Java garbage collector always runs in a separate thread, even the simplest Java program is actually multithreaded.

In the previous chapters we saw that Java applications begin when the JVM invokes the `main()` method. (The application itself runs as a thread in the JVM.) Instead of a `main()`, the thread processes begin and end with a method named `run()`. You place code in `run()` to control the operations that you wish to accomplish with the thread. The thread lives only as long as the process remains within `run()`. When the thread process returns from the `run()`, the thread is dead and cannot be resurrected.

You create a thread class in one of two ways:

1. Create a subclass of the `Thread` class and override the `run()` method.
2. Create a class that implements the `Runnable` interface, which has only one method: `run()`. Pass a reference to this class in the constructor of `Thread`. The thread then calls back to this `run()` method when the thread starts.

In the following sections we examine these two thread creation techniques further.

8.2.1 Thread creation: subclass

Creating a subclass of `Thread` offers the most conceptually straightforward approach to threading. In this approach the subclass overrides the `run()` method with the code you wish to process. The following code segments illustrate this approach.

The class `MyThread` extends the `Thread` class and overrides the method `run()` with one that contains a loop that prints out a message until a counter hits 20.

```
public class MyThread extends Thread
{
   public void run () {
     int count = 0;
     while (true) {

         System.out.println ("Thread alive");

         // Print every 0.10sec for 2 seconds
         try {
           Thread.sleep (100);
         }
         catch (InterruptedException e) {}
         count++;
         if (count >= 20) break;
     }
     System.out.println ("Thread stopping");
   } // run
} // class MyThread
```

In `MyApplet` shown below, the `start()` method creates an instance of the `MyThread` class and invokes the thread's `start()` method. This will in turn invoke the `run()` method. The thread goes into a loop and prints a message every 100 ms using the `Thread` class static method `sleep(long time)`, where time is in milliseconds. The thread then dies – i.e. it cannot be restarted – once the process exits from `run()`.

```
/** Demo threading with Runnable implementation.**/
public class MyApplet extends java.applet.Applet
{
  /** Applet's start method creates and starts a thread.
    **/
  public void start () {

    // Create an instance of MyThread.
    MyThread myThread = new MyThread ();

    // Start the thread
    myThread.start ();
  } // start

  public void paint(java.awt.Graphics g) {
    g.drawString("Thread Demo 1",20,20);
  }
} // class MyApplet
```

The diagram in Figure 8.1 shows schematically how the main thread and `MyThread` thread run in parallel.

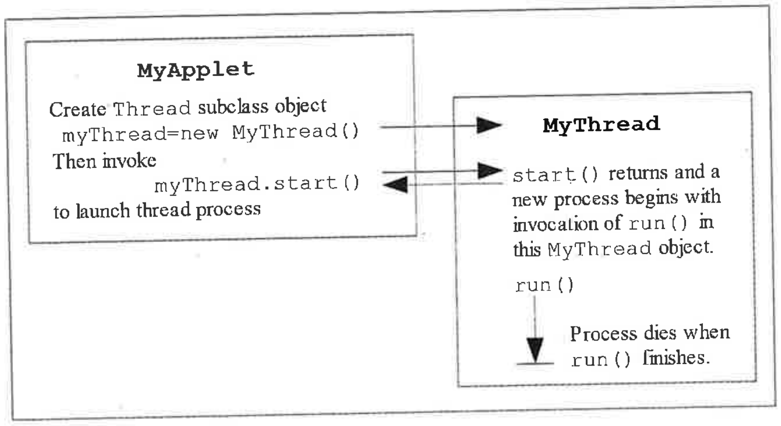

MyApplet
Create Thread subclass object
myThread=new MyThread()
Then invoke
myThread.start()
to launch thread process
MyThread
start() returns and a new process begins with invocation of run() in this MyThread object.
run()
Process dies when run() finishes.

Figure 8.1 This diagram illustrates threading with a `Thread` subclass. `MyApplet` creates an instance of `MyThread`, which extends `Thread` and overrides `run()`. The applet invokes the `start()` method for the `MyThread` object. The `start()` method returns but the thread process continues independently with the invocation of the `run()` method in `MyThread`. When the thread process returns from `run()` the thread dies (i.e. cannot be started again).

8.2.2 Thread creation: Runnable

In the second threading technique a class implements the `Runnable` interface and overrides its `run()` method. This approach is often convenient, especially for cases where you want to create a single instance of a thread, as in an animation for an applet. You pass a reference to the `Runnable` object via the constructor of `Thread` and when it starts, the thread *calls back* to the `run()` method. As before, the thread process dies after exiting `run()`.

The following code segment illustrates this approach. Here `MyRunnableApplet` implements the `Runnable` interface. The `start()` method creates an instance of the `Thread` class and passes a reference to itself (with the "`this`" reference) in the thread's constructor. When it invokes the `start()` method for the thread, the thread will invoke the `run()` method in `MyRunnableApplet`.

```
/** Demo threading with Runnable implementation. **/
public class MyRunnableApplet extends java.applet.Applet
                              implements Runnable
{
  /** Applet's start method creates a thread. **/
  public void start () {

    // Create an instance of Thread with a
    // reference to this Runnable instance.
    Thread thread = new Thread (this);

    // Start the thread
    thread.start ();
  } // start

  /** Override the Runnable run() method. **/
  public void run () {
    int count = 0;
    while (true) {
      System.out.println ("Thread alive");
      // Print every 0.10sec for 5 seconds
      try{
        Thread.sleep(100);
      } catch (InterruptedException e) {}
      count++;
      if (count >= 50) break;
    }
    System.out.println ("Thread stopping");
  } // run

  public void paint (java.awt.Graphics g) {
      g.drawString ("Thread demo 2",20,20);
    }
  } // class MyRunnableApplet
```

The diagram in Figure 8.2 shows schematically how this approach works.

8.2.3 `Thread` subclass vs. `Runnable`

The choice between these two thread creation techniques depends on the particular application and what seems most appropriate and convenient for it. Since Java does not allow multiple inheritance, an applet class that is already a subclass of `Applet` or `JApplet` can become multithreaded by implementing `Runnable`. The `run()` method will have access to the variables and methods of the class. For example, an applet animation may need parameters for initialization and may also need to invoke methods from the applet.

Extending `Thread` applies well to the situation where you want to create a specialized thread class that does not need to extend any other class. A common case is where many *worker* threads are needed such as in a server program that

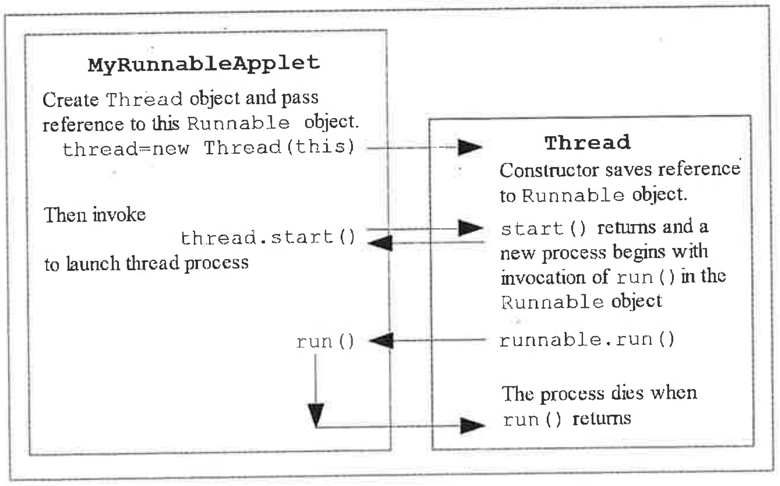

MyRunnableApplet
Create Thread object and pass reference to this Runnable object.
thread=new Thread(this)
Then invoke
thread.start()
to launch thread process
run()
Thread
Constructor saves reference to Runnable object.
start() returns and a new process begins with invocation of run() in the Runnable object
runnable.run()
The process dies when run() returns

Figure 8.2 This diagram illustrates threading with a `Runnable` class. `MyRunnableApplet` implements the `Runnable` interface and it creates an instance of `Thread` and passes in the constructor a reference to itself as a `Runnable` object. (We use the name "`runnable`" for the reference variable to the `Runnable` object.) The applet invokes the `start()` method for the thread and it returns while the thread process continues independently with the invocation of the `run()` method *in* the applet object. When the thread process returns from the applet's `run()` the thread dies.

assigns a worker to service each client that connects to it. When the client signs off, the threaded worker process assigned to it dies.

8.3 Stopping threads

A thread dies in three ways:

- it returns from `run()`
- the `stop()` method is invoked (this method is now deprecated)
- it is interrupted by a runtime exception

The first approach is always the preferred way for a thread to die. In the examples shown above in Section 8.2, we used a flag variable in a loop to tell the run method to finish. We recommend this approach to killing a thread.

Do *not* use the `Thread` method `stop()` to kill a thread. The `stop()` method has been *deprecated*. That means that it still exists in the class definition but is officially marked as obsolete and should be avoided. The `stop()` method causes the thread to cease whatever it is doing and to throw a `ThreadDeath` exception. This can leave data in an unknown state. For example, the thread might be midway through setting the values of a group of variables when the thread was stopped. This will leave some variables with new values and some with old values. Other processes using those variables might then obtain invalid results. Furthermore, an instruction involving a `long` or `double` type value can require two operations in the JVM, which moves data in 32-bit chunks. So a thread stop might occur after the first operation and leave the value in an indeterminate state. These kinds of errors will be difficult to track down since the effect may not be seen until the processing reaches another part of the program.

As mentioned earlier, the best way to stop a thread is to signal that the processing should return from `run()`. Setting a flag can usually accomplish this. A loop can check the flag after each pass and escape from the loop with the flag switches. This allows for the process to finish what it is doing in a controlled manner. In previous examples we set a `boolean` flag. In the applet below we use the thread reference instead of a separate flag variable. Setting the reference to `null` signals for the end of a loop in `run()` and also allows the garbage collector to reclaim the memory used by the thread.

```
public class MyApplet extends Applet implements Runnable
{
  Thread fMyThread;
  public void init () {
    . . .
```

```
  }
  . . .
  publi
    if
  }
    els
  }
  publi
    fMy
  }
  void
    whi
    }
  }
} // My
```

Remember
unrelated tc
method in
control the
loaded (not
loaded). Th
any live thr
`stop()` is
loads a new
Furtherm
in the `Thre`
You can obt
signaling fc
creating a r
thread died
finished.

8.4 Mul

An operatin
for multithr

```
    }
    . . .
    public void start () {
       if (fMyThread!= null) {
          fMyThread.start ();
       }
       else
          fMyThread = new Thread (this);
    }

    public void stop () {
       fMyThread = null;
    }

    void run () {
       while (fMyThread!= null) {
          . . .
       }
    }
  } // MyApplet
```

Remember that the `start()` and `stop()` methods in the `Applet` class are unrelated to methods with the same names in the `Thread` class. Like the `init()` method in the `Applet` class, these are just methods that allow the browser to control the applet. The browser invokes `start()` each time the applet page is loaded (note that `init()` is only invoked the first time the applet web page is loaded). The applet's `stop()` is a good place to do housecleaning such as killing any live threads. Always explicitly stop your threads in applets when the applet `stop()` is called. Otherwise, they may continue running even when the browser loads a new web page.

Furthermore, do *not* use the deprecated `suspend()` and `resume()` methods in the `Thread` class for the same reasons given for not using the `stop()` method. You can obtain effective suspend/resume operations by killing the thread (that is, signaling for it to return safely from the processing in the `run()` method) and creating a new one with the same values of the variables as when the previous thread died. The new thread will then simply continue from where the last one finished.

8.4 Multiprocessing issues

An operating system executes multiple processes in a manner similar to that for multithreading except that each process stack refers to a different program in memory rather than code within a single program. The Java Virtual Machine (JVM) controls the threads within a Java program much as the machine operating system controls multiple processes.

In some JVM implementations, threads are directly assigned to native processes in the operating system. Furthermore, in operating systems that support multiple physical processors, the threads can actually run on different processors and thus achieve true parallel processing.

Multiprocessing in Java with threads is relatively straightforward and provides for great flexibility in program design. The JVM handles most of the details of running threads but your program can influence the sharing of resources and setting priorities for multiple threads.

8.4.1 Sharing resources

Just as in an operating system, when multiple threads need to share a processor or other resources, the JVM must provide a mechanism for a thread to pause and allow other threads the opportunity to run. The two basic designs for *context* switching of threads are:

- *preemptive* or *time-slicing* – give each thread fixed periods of time to run
- *non-preemptive* or *cooperative* – a thread decides for itself when to surrender control

Generally, the preemptive approach is the most flexible and robust. A misbehaving thread cannot hog all the resources and possibly freeze the whole program. Unfortunately, the context switching design is not specified currently for Java and so different JVMs do it differently. Thus you should design your multithreaded code for either possibility if you expect to distribute your program for general use.

For example, you can explicitly add pauses to your threads to ensure they share the processing. The static method `yield()` in the `Thread` class tells the currently executing thread to pause momentarily and let other threads run. The static method `sleep(long millis)`, where `millis` is in milliseconds, tells the currently executing thread to pause for a specific length of time. There is also the overloaded version method `sleep(long millis, int nanos)`, where the sleep time equals `millis` in milliseconds plus `nanos` in nanoseconds. (Most platforms, however, cannot specify time that accurately.) With these two methods, you can ensure that when your program runs in a non-preemptive environment, the threads will release control at suitable points in the thread code.

The resources needed for each thread is another aspect of multiprocessing to consider when creating a high number of threads. The maximum number of threads depends on the stack space needed per thread and the amount of memory available. The stack size default is 400 KB. For a 1 gigabyte address space this

should allow
itself plus an
an `OutOfMe`

8.4.2 Set

Every thread
trolled using
can be expec
ority threads.
the JVM imp
ity to JVM d
speed and res
resources.
The JVM i
ing capabiliti
Even among
details of tha
and perhaps a
certain is tha
the thread sc
more threads
take into acc
thread is per
Over a long
scheduled m
at any given
a higher prio
not a reliable
another. (See
which expan
of thread exe
With tho
`getPriori`
`Thread clas`

- `MIN_PRIO`
- `NORM_PRI`
- `MAX_PRIO`

The default
threads alwa

should allow up to 2500 tiny threads, but in practice, because the thread code itself plus any memory allocated for objects a thread uses takes up memory too, an `OutOfMemoryError` will usually occur far sooner.

8.4.2 Setting priorities

Every thread has an integer priority value between 1 and 10 that can be controlled using methods in the `Thread` class. Generally, higher priority threads can be expected to be given preference by the thread scheduler over lower priority threads. However, the implementation of thread scheduling is left up to the JVM implementation. This lack of specificity provides maximum flexibility to JVM designers since Java can be implemented on platforms with limited speed and resources and also on platforms with multiple processors and extensive resources.

The JVM implementation must work within the native platform's multithreading capabilities, which might or might not include native multithreading features. Even among host operating systems that natively support multiple threads, the details of that support are sure to be different among different operating systems and perhaps among different hardware platforms. About all that can be said for certain is that higher priority threads *should* receive preferential treatment by the thread scheduler compared to threads with lower priority. However, if two or more threads are waiting for processor resources, the thread scheduler may also take into account how long the threads have been waiting. The highest priority thread is perhaps likely to be the first to be scheduled, though not necessarily. Over a long enough sampling time, higher priority threads will, on average, be scheduled more often than lower priority threads, but that does not mean that at any given time a lower priority thread might have control of the CPU while a higher priority thread is waiting. In general, changing Java thread priorities is not a reliable way to attempt to force one thread to *always* have preference over another. (See Section 24.4 for a discussion of the real-time specification for Java, which expands the number of priority levels to 28 and requires strict enforcement of thread execution according to priority settings.)

With those caveats, you can get and set a thread's priority with the `getPriority()` and `setPriority()` methods in the `Thread` class. The `Thread` class defines three constants:

- `MIN_PRIORITY`
- `NORM_PRIORITY`
- `MAX_PRIORITY`

The default priority is `Thread.NORM_PRIORITY`, which is 5, although new threads always inherit the priority value of the creating thread. The following code increments a thread's priority to one unit higher than the normal priority:

```
  . . .
  Thread threadX = new Thread (this);
  threadX.setPriority (Thread.NORM_PRIORITY + 1);
  threadX.start ();
  . . .
```

Attempting to set a thread's priority below `MIN_PRIORITY` or above `MAX_PRIORITY` results in an `IllegalArgumentException`.

For multiple threads in a non-preemptive system, once one of them starts running it will continue until one of the following happens:

- sleeps via an invocation of `sleep()`
- yields control with `yield()`
- waits for a lock in a synchronized method (synchronization is discussed in the next section)
- blocks on I/O such as a `read()` method waiting for data to appear
- terminates with a return from `run()`

We will discuss synchronization and the `wait()` method in the following section.

8.5 Using multiple threads

Programs for some tasks become much easier to design with threads, sometimes with lots of threads. We've already mentioned animations, and in Part II we will see that client/server systems lend themselves naturally to multithreaded design – the server can spin off a new thread to service each client. Some mathematical algorithms, such as sorting and prime searching, also work well with multiple threads working on different segments of the problem. On multiprocessor systems, JVMs can take advantage of true parallel processing and provide significant speedups in performance for multithreaded applications.

There are basically four situations in which multiple threads operate:

1. **Non-interacting threads** – the actions of the threads are completely independent and do not affect each other.
2. **Task splitting** – each thread works on a separate part of the same problem, such as sorting different sections of a data set, but do not overlap with each other.
3. **Exclusive thread operations** – the threads work on the same data and must avoid interfering with each other. This requires *synchronization* techniques.
4. **Communicating threads** – the threads must pass data to each other and do it in the correct order.

The latter two cases can entail complex and often subtle interference problems among the threads. We look in more detail at these four cases in the following sections.

8.5.1 N

The simple
dently with
such a case
the values
for demons

```
/** Dem
  * cal
class I
{
  int f
  int f
  int f
  Outpu

/** Con
IntCoun
  fId =
  fMaxI
  fOutp
} // ct

/** Sim
public
  while
  fOutp
  }
} // cl
```

The progra
interface di
tons (see Fi
which creat

```
import
import
import

public

{
  . . .
```

8.5.1 Non-interacting threads

The simplest situation with multiple threads is when each thread runs independently without interacting with any other thread. Below is a simple example of such a case. We have one `Thread` subclass called `IntCounter` that prints out the values of an integer counter. We could do a more interesting calculation but for demonstration purposes this will suffice.

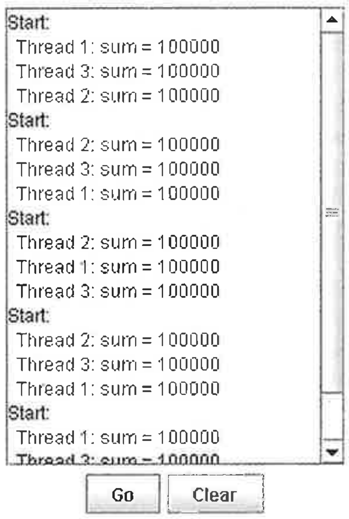
```
/** Demo Thread class to show how threads could do
  * calculations in parallel. **/
class IntCounter extends Thread
{
  int fId=0;
  int fCounter = 0;
  int fMaxIter = 0;
  Outputable fOutput;

/** Constructor to initialize parameters. **/
IntCounter (int id, Outputable out) {
  fId = id;
  fMaxIter = 100000
  fOutput = out;
} // ctor

/** Simulate a calculation with an integer sum.**/
public void run () {
  while (fCounter < maxIter) fCounter++;
  fOutput.println ("Thread" + fId + ": sum = " + fCounter);
  }
} // class IntCounter
```

The program `NonInteractApplet`, which implements the `Outputable` interface discussed in Chapter 6, provides a text area and "Go" and "Clear" buttons (see Figure 8.3). Clicking on "Go" invokes the applet's `start()` method, which creates three instances of this `Thread` subclass and starts them:

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class NonInteractApplet extends JApplet
                     implements Outputable, ActionListener
{
  . . . Build interface . . .
// Pushing Go button leads to the invocation of this button
public void start () {
   // Create 3 instances of each of the Thread subclass
   IntCounter ic1 = new IntCounter (1, this);
   IntCounter ic2 = new IntCounter (2, this);
   IntCounter ic3 = new IntCounter (3, this);

   // Start the threads
   ic1.start ();
   ic2.start ();
   ic3.start ();
   } // start
} // class NonInteractApplet
```

In the output shown in Figure 8.3 you see that the threads can finish in a different order for each press of "Go". The order depends on the time allocated to each thread and on what kind of thread scheduling the JVM uses. You can experiment with different thread priorities by adding code to set the priorities of the three thread instances differently. For instance, set `ic1` to a high priority and `ic3` to a low priority before starting the threads

```
ic1.setPriority (Thread.MAX_PRIORITY);
ic2.setPriority (Thread.NORM_PRIORITY);
ic3.setPriority (Thread.MIN_PRIORITY);
```



```
Start:
 Thread 1: sum = 100000
 Thread 3: sum = 100000
 Thread 2: sum = 100000
Start:
 Thread 2: sum = 100000
 Thread 3: sum = 100000
 Thread 1: sum = 100000
Start:
 Thread 2: sum = 100000
 Thread 1: sum = 100000
 Thread 3: sum = 100000
Start:
 Thread 2: sum = 100000
 Thread 3: sum = 100000
 Thread 1: sum = 100000
Start:
 Thread 1: sum = 100000
 Thread 3: sum = 100000
```

Figure 8.3 Display of the `NonInteractApplet` program. The Go button has been pushed several times to illustrate the different order in which the threads finish.

8.5.2 Ta

The next le
problem but
integer valu
using differ
the specified
In the ex
snippet bel
positive non

```
/**
 * Thre
 * in a
 **/
class Ma
{
  int [
  int f
  int f
  Output

/** Cons
 * what
 **/
MatHunte
  int [
  int i
  Output
 ) {
  fIlo=
  fJlo=
  fMatr
  fOutp
  } //

  /** E
   * c
   **/
  publi
   for
    f
```

button
ss

a different
ed to each
:xperiment
f the three
d ic3 to a

8.5.2 Task splitting

The next level in complexity involves multiple threads working on the same problem but on separate, non-interfering parts. For example, given a particular integer value, a program could find the number of primes up to that value by using different threads to work on different sections of the range between 1 and the specified value.

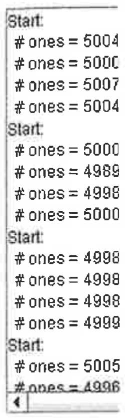
In the example here, we use the task-splitting technique to scan a matrix. The snippet below shows a class that searches a matrix and counts the number of positive non-zero elements:

```
/**
  * Thread class to count the number of non-zero elements
  * in a section of a matrix.
 **/
class MatHunter extends Thread
{
   int [][] fMatrix;
   int fIlo, fIhi, fJlo, fJhi;
   int fOnes=0;
   Outputable fOutput;

/** Constructor gets the matrix and the indices specifying
  * what section to examine.
 **/
MatHunter (
   int [][] imat,
   int i1, int i2, int j1, int j2,
   Outputable out
) {
   fIlo=i1; fIhi=i2;
   fJlo=j1; fJhi=j2;
   fMatrix = imat;
   fOutput = out;
   } // ctor

   /** Examine a section of a 2D matrix and
     * count the number of non-zero elements.
    **/
   public void run () {
      for (int i=fIlo; i <= fIhi; i++) {
         for (int j=fJlo; j <= fJhi; j++) {
            if (fMatrix[i][j] > 0) fOnes++;
         }
         yield ();
      }
      fOutput.println ("# ones =" + fOnes + "for i =" +
        fIlo + "to" + fIhi + "& j =" + fJlo + "to" + fJhi);
    } // run
  } // class MatHunter
```

The program `TaskSplitApplet` creates a matrix with a random distribution of zero and non-zero elements. It then creates four instances of `MatHunter`, one for each quadrant of the matrix. Each instance works on the same problem but in a separate, independent section of the matrix.

```
public class TaskSplitApplet extends JApplet
           implements Outputable, ActionListener
{
  . . . Build the interface . . .

  public void start () {
    int[][] imat = new int[2000][2000];

    for (int i=0; i < 2000; i++) {
      for (int j=0; j < 2000; j++) {
        if (Math.random() > 0.5) imat[i][j] = 1;
      }
    }
    MatHunter mh1 = new MatHunter (imat,0,999,0,999,this);
    MatHunter mh2 =
      new MatHunter (imat,0,999,1000,1999,this);
    MatHunter mh3 =
      new MatHunter (imat,1000,1999,0,999,this);
    MatHunter mh4 =
      new MatHunter (imat,1000,1999,1000,1999,this);

    Println ("Start:");
    mh1.start ();
    mh2.start ();
    mh3.start ();
    mh4.start ();
  } // start
} // class TaskSplitApplet
```

Figure 8.4 shows the results of different threads finishing in a different order each time the "Go" button is pressed.



```
Start:
 # ones = 5004
 # ones = 5000
 # ones = 5007
 # ones = 5004
Start:
 # ones = 5000
 # ones = 4989
 # ones = 4998
 # ones = 5000
Start:
 # ones = 4998
 # ones = 4998
 # ones = 4998
 # ones = 4999
Start:
 # ones = 5005
 # ones = 4996
```

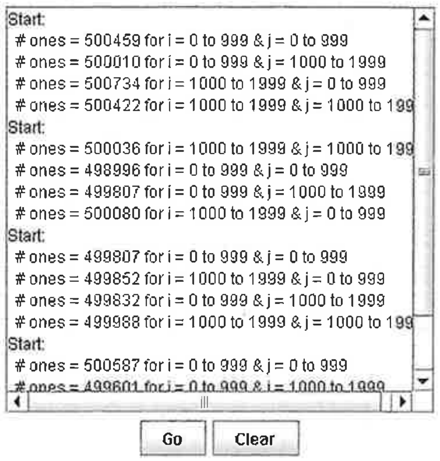
Figure 8.4 D
result in a dif

8.5.3 Ex

Threading b
with each o
processes bo
from an exa
a number in
ity is empty
from Cavit
would altern
special steps
the Cavity
still full.

This type
do its task w
nization sch
in single file
occur.

In this ca
to invoke eit
lock on the
other thread
lock termino
The term *mo*



```
Start:
 # ones = 500459 for i = 0 to 999 & j = 0 to 999
 # ones = 500010 for i = 0 to 999 & j = 1000 to 1999
 # ones = 500734 for i = 1000 to 1999 & j = 0 to 999
 # ones = 500422 for i = 1000 to 1999 & j = 1000 to 199
Start:
 # ones = 500036 for i = 1000 to 1999 & j = 1000 to 199
 # ones = 498996 for i = 0 to 999 & j = 0 to 999
 # ones = 499807 for i = 0 to 999 & j = 1000 to 1999
 # ones = 500080 for i = 1000 to 1999 & j = 0 to 999
Start:
 # ones = 499807 for i = 0 to 999 & j = 0 to 999
 # ones = 499852 for i = 1000 to 1999 & j = 0 to 999
 # ones = 499832 for i = 0 to 999 & j = 1000 to 1999
 # ones = 499988 for i = 1000 to 1999 & j = 1000 to 199
Start:
 # ones = 500587 for i = 0 to 999 & j = 0 to 999
 # ones = 499601 for i = 0 to 999 & j = 1000 to 1999
```

Figure 8.4 Display of `TaskSplitApplet` program. Pressing the `"Go"` button can result in a different sequence in the completion times of the thread each time.

8.5.3 Exclusive thread operations

Threading becomes trickier when threads perform operations that can conflict with each other. For example, Figure 8.5 depicts a situation where two thread processes both want to access an object but for different purposes (this is derived from an example in the Sun Java Tutorial). The `Filler` thread wants to put a number into the `bin` variable in the `Box`. It can only do so when the cavity is empty. The `Getter`, on the other hand, wants to retrieve the number from `Cavity` and leave the `Cavity` empty. Ideally, `Filler` and `Getter` would alternate their calls to the methods `put()` and `get()`. However, if no special steps are taken, it is quite easy for `Getter` to invoke `get()` when the `Cavity` is empty and for `Filler` to invoke `put()` when the `Cavity` is still full.

This type of situation is called a *data race* because each thread is racing to do its task without waiting for the other thread to finish its activity. A *synchronization* scheme prevents this problem. Synchronization forces threads to wait in single file at the method or code block of an object where the conflict can occur.

In this case, this means that the `Box` object only allows one thread at a time to invoke either its `put()` or `get()`. It is as if only one thread object owns the *lock* on the door to a `Box` object. That thread must give up the lock before any other thread can access *any* synchronized method *on the object*. (Note that the *lock* terminology is by convention. Giving up the *key* might be more illuminating. The term *monitor* is also used.)

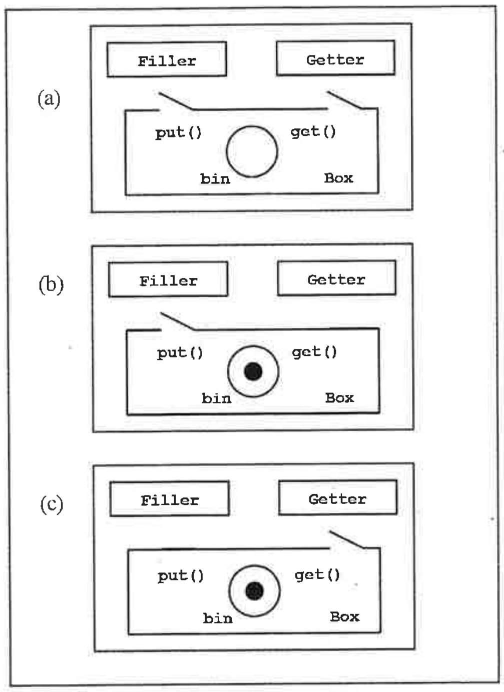


Figure 8.5 (a) The `Filler` and `Getter` threads need to access the bin in the `Box`. The `Getter` needs, however, to wait till the bin is filled. (b) While the `Filler` places a value in the bin via the synchronized `put()` method, the `Getter` cannot invoke the synchronized `get()` method. (c) Similarly, the `Filler` must wait till the `Getter` finishes invoking the `get()` method before it can invoke `put()`.

In the following code for the `Box` class, we see that the `get()` and `put()` methods are prefaced by the modifier `synchronized`. This indicates that only one thread can invoke either of these methods at the same time for the same object. That is, during the time that a thread executes, say, the `get()` method, no other thread can execute either the `get()` or `put()` method for the same `Box` object.

```
public class Box
{
  private int fBin;
  private boolean fFilled = false;
  Outputable fOutput;
  /** Constructor obtains reference to Outputable object.
    **/
  Box (Outputable out) {
    fOutput = out;
```

```
} // 

/** I
publi
  whi
    t

    }
    c
  }
  fFi
  fOu
  not
  ret
} // g

/** If
public
  whil
    t

    }
    ca
  }
  fBin
  fFil
  fOut
  noti
} // p
} // clas
```

We want to e
interference
one Box obje
different Box
This code
thread invoke
object becaus
is granted the
check is made
already true (
Similarly, dur
wait() is in
and remains a

ı the Box.
.ler places a
invoke the
Getter

and put()
tes that only
same object.
ıod, no other
Box object.

ɔbject.

```
  } // ctor

  /** If bin is not filled, wait for it to be. **/
  public synchronized int get () {
    while (!fFilled){
      try {
        wait ();
      }
      catch (InterruptedException e) {}
    }
    fFilled = false;
    fOutput.println ("Get value:" + fBin);
    notifyAll ();
    return fBin;
  } // get

  /** If bin is filled, wait for it to be emptied. **/
  public synchronized void put (int value){
    while (fFilled) {
      try {
        wait ();
      }
      catch (InterruptedException e) {}
    }
    fBin = value;
    fFilled = true;
    fOutput.println ("Put value: " + fBin);
    notifyAll ();
  } // put
} // class Box
```

We want to emphasize that each instance of Box has its own lock. There is no interference problem among different Box objects. If a thread owns the lock on one Box object, this does not prevent another thread from owning the lock on a different Box object.

This code also illustrates the wait() and notifyAll() methods. When a thread invokes put() or get(), it will wait until it is granted the lock for that object because of the presence of the synchronized keyword. Once the thread is granted the lock, it continues on through the method. Inside the method, a check is made on the fFilled flag. When attempting a put(), if fFilled is already true (i.e. if the bin is already full), then an explicit wait() is invoked. Similarly, during a get(), if fFilled is false (i.e. if the bin is empty), then wait() is invoked. Invoking wait() means that the thread gives up the lock and remains at that point in the method until notified to continue.

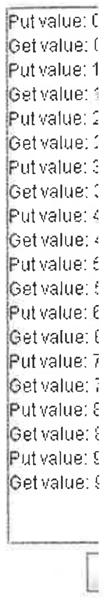
Let's suppose that the `Filler` thread finds that `fFilled = true` during the `put()` method; that thread will go into a wait state. Since `fFilled` is true, the `Getter` thread passes the `fFilled` test in the `get()` method, obtains the `fBin` value, sets the `fFilled` flag to `false`, and invokes `notifyAll()` before it returns. The `notifyAll()` method causes all threads in a wait state to attempt to acquire the lock. When the lock is released by the `Getter` in the synchronized `get()` method, the `Filler` thread can acquire the lock and continue on through the `put()` method and fill the bin again.

The following code shows the `Filler` class. In the `run()` method, a loop puts a value into the box and then pauses for a random period of time before doing it again. For each pass of the loop, the `put()` invocation results in the printing of a message via the `Outputable` reference.

```
public class Filler extends Thread
{
  private Box fBox;

  public Filler (Box b) {
    fBox = b;
  }

  public void run () {
    for (int i=0; i < 10; i++) {
      fBox.put (i);
      try {
        sleep ((int)(Math.random () * 100));
      }
      catch (InterruptedException e) {}
    }
  } // run
} // class Filler
```

The following code shows the `Getter` class. The loop in its `run()` method will continue until it gets ten values from the box. Note, however, that the process will experience occasional wait states in the `get()` method in `Box` to give time for the `Filler` to do its job.

```
public class Getter extends Thread
{
  private Box fBox;
  private int fNumber;

  public Getter (Box b) {
    fBox = b;
```



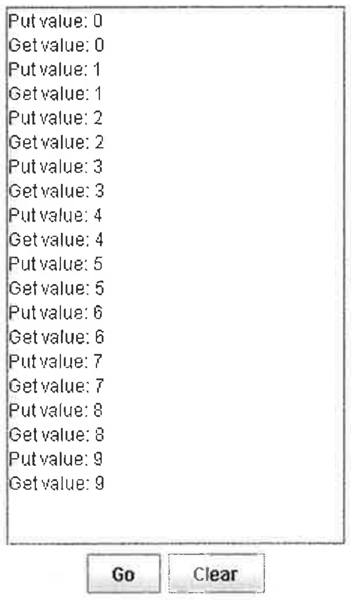
```
Put value: 0
Get value: (
Put value: 1
Get value: 1
Put value: 2
Get value: 2
Put value: 3
Get value: 3
Put value: 4
Get value: 4
Put value: 5
Get value: 5
Put value: 6
Get value: 6
Put value: 7
Get value: 7
Put value: 8
Get value: 8
Put value: 9
Get value: 9
```

Figure 8.6
as they fill

```
  }
  publ
    ir
    fc

    }
  } //
} // c
```

The snipp
and a Get
output. W
threads ea

```
public
{
```



```
Put value: 0
Get value: 0
Put value: 1
Get value: 1
Put value: 2
Get value: 2
Put value: 3
Get value: 3
Put value: 4
Get value: 4
Put value: 5
Get value: 5
Put value: 6
Get value: 6
Put value: 7
Get value: 7
Put value: 8
Get value: 8
Put value: 9
Get value: 9
```

Figure 8.6 The output of the `Filler` and `Getter` threads for the `ExclusiveApplet` as they fill and retrieve a bin value in a `Box` object.

```
    }
    public void run () {
      int value = 0;
      for (int i=0; i < 10; i++) {
          fNumber = fBox.get ();
      }
    } // run
  } // class Getter
```

The snippet from `ExclusiveApplet` shown below creates a `Box`, a `Filler`, and a `Getter` object and then starts the two threads. Figure 8.6 shows a typical output. We see that the synchronization prevents a data race situation and the two threads each complete their respective tasks.

```
public class ExclusiveApplet extends JApplet
            implements Outputable, ActionListener
{
  . . . Build the interface . . .
```

```
    /** Create Filler and Getter thread instances and start
      * them filling and getting from a Box instance. **/
    public void start () {

      Box b = new Box (this);
      Filler f1 = new Filler (b);
      Getter b1 = new Getter (b);
      f1.start ();
      b1.start ();
    } // start
    . . .
  } // class ExclusiveApplet
```

8.5.4 Communications among threads

In the previous section, we discussed the case where multiple threads try to access an object and can step on each other if not properly synchronized. Here we look at the even trickier situation where a thread needs to access data in another thread and must also avoid a data race situation.

The standard example for communicating threads is the *producer/consumer* paradigm. The producer object invokes its own synchronized method to create the data of interest. The consumer cannot invoke the producer's `get()` method, which is also synchronized, until the producer has finished with its creation method. The producer, in effect, locks its own door to the consumer until it finishes making the data. (Imagine a physical store that locks its doors and does not allow shoppers in while restocking the shelves.) Similarly, while the consumer gets the data from the producer, it obtains the *lock* and prevents the producer from generating more data until the consumer is finished.

Below we illustrate this paradigm with a program in which the `Sensor` class represents the producer thread and `DataGetter` represents the consumer thread. An instance of `Sensor` obtains its data (here just clock readings) in a loop in `run()` via calls to the synchronized `sense()` method. The data goes into a buffer array. A thread can invoke `get()` in `Sensor` to obtain the oldest data in the buffer. The indices are set up to emulate a FIFO (First-In-First-Out) buffer. When the buffer is full, the `Sensor` thread waits for data to be read out (that is, it gives up the lock by calling the `wait()` method).

To obtain the data, a `DataGetter` instance invokes the synchronized `get()` method in the `Sensor` instance. If no new data is available, it will give up the lock and wait for new data to appear (that is, when `notifyAll()` is invoked in the `sense()` method).

This snippet from `DataSyncApplet` creates the sensor and starts it. Then a `DataGetter` is created and started.

```
public class DataSyncApplet extends JApplet
            implements Outputable, ActionListener
{
  . . . Build the interface . . .

  /** Create Sensor and DataGetter thread instances and
    * start them filling and getting from a Box instance.
   **/
  public void start() {
    // Create the Sensor and start it
    Sensor s = new Sensor (this);
    s.start ();

    // Create DataGetter and tell it to obtain
    // 100 sensor readings.
    DataGetter dg = new DataGetter (s, 100, this);
    dg.start ();
  } // start
  . . .
} // class DataSyncApplet
```

The `Sensor` (see code below) produces one data value (just a string containing the number of milliseconds since the program began) and stores it in an element of a buffer array. The `fBufIndex` keeps track of where the next value should go. When it reaches the end of the array, it will circle back to the start. The `fGetIndex` marks the value in the buffer that will be sent next to the `DataGetter`. The `fGetIndex` should never fall farther behind `fBufIndex` than the `MAXGAP` value (set here to 8). If the lag reaches the value of `fMaxGap` then the sensor goes into a loop with an invocation of `wait()` for each pass. When the `DataGetter` invokes the `get()` method, the `notifyAll()` will wake the `Sensor` thread from its wait state and it will check the lag again. If it is no longer at the maximum, the process leaves the wait loop and produces more data. Otherwise, it loops back around and invokes `wait()` again.

```
import java.util.*;
/**
 * This class represents a sensor producing data
 * that the DataGetter objects want to read.
 */
public class Sensor extends Thread
{
  // Size of the data buffer.
  private static final int BUFFER_SIZE = 10;
```

```
// Don't let data production get more than
// 8 values ahead of the DataGetter
private static final int MAXGAP = 8;
private String [] fBuffer;
private int fBufIndex = 0; // sensor data buffer index
private int fGetIndex = 0; // data reading index
private final long fStart = System.currentTimeMillis ();

boolean fFlag = true;
Outputable fOutput;

/** Constructor creates buffer. Gets Outputable ref. **/
Sensor (Outputable out) {
  fOutput = out;
  fBuffer = new String [BUFFER_SIZE];
}
/** Turn off sensor readings. **/
public void stopData () {
  fFlag = false;
}
/** Take sensor readings in a loop until flag set false.
 **/
public void run () {
  // Measure the parameter of interest
  while (fFlag) sense ();
}

/** Use clock readings to simulate data. **/
private final String simulateData () {
    return "" + (int) (System.currentTimeMillis () -
    start);
}

/** Use indices fBufIndex, fGetIndex, and the lag()
  * method to implement a first-in-first-out (FIFO)
  * buffer. **/
synchronized void sense () {
  // Don't add more to the data buffer until the getIndex
  // has reached within the allow range of bufIndex.
  while (lag () > MAXGAP) {
    try {wait ();}
    catch (Exception e) {}
  }
  fBuffer[fBufIndex] = simulateData ();
  fOutput.println("Sensor["+ (fBufIndex) + "] = "
                         + fBuffer[fBufIndex]);
```

```
      // Increment index to next slot for new data
      fBufIndex++;
      // Circle back to bottom of array if reaches top
      if (fBufIndex == BUFFER_SIZE) fBufIndex = 0;
      notifyAll ();
    } // sense

    /** Calculate distance the DataGetter is running behind
      * the production of data. **/
    int lag () {
      int dif = fBufIndex - fGetIndex;
      if (dif < 0) dif += BUFFER_SIZE;
      return dif;
    }
    /** Get a data reading from the buffer. **/
    synchronized String get () {
      // When indices are equal, wait for new data.
      while (fBufIndex == fGetIndex) {
        try{ wait(); }
        catch (Exception e) {}
      }
      notifyAll ();

      // Get data at current index
      String data = fBuffer[fGetIndex];

      // Increment pointer of next datum to get.
      fGetIndex++;
      // Circle back to bottom of array if reaches top
      if (fGetIndex == BUFFER_SIZE) fGetIndex = 0;
      return data;
    } // get
  } // class Sensor
```

The `DateGetter` grabs a data value from the sensor after random delay until it gets its maximum number of data values. Figure 8.7 shows typical output from `DataSync`.

```
import java.util.*;

/** This class obtains sensor data via the get () method.
  * To simulate random accesses to the sensor, it will
  * sleep for brief periods of different lengths after
  * every access. After the data is obtained, this thread
  * will stop the sensor thread. **/
public class DataGetter extends Thread
{
  Sensor fSensor;
  Outputable fOutput;
  int fMaxData = 1000;
  int fDataCount = 0;

  DataGetter (Sensor sensor, int maxNum, Outputable out) {
    fSensor = sensor;
    fMaxData = maxNum;
    fOutput = out;
  }
  /** Loop over sensor readings until data buff filled. **/
  public void run () {
    Random r = new Random ();
    while (true) {
      String data = fSensor.get();
      fOutput.println(fDataCount++ + ". Got: " + data);

      // Stop both threads if data taking finished.
      if (fDataCount >= fMaxData) {
        fSensor.stopData ();
        break;
      }
      // Pause briefly before access the
      // data again.
      try {
        sleep (r.nextInt () % 300);
      }
      catch (Exception e) {}
    }
  } // run
} // class DataGetter
```

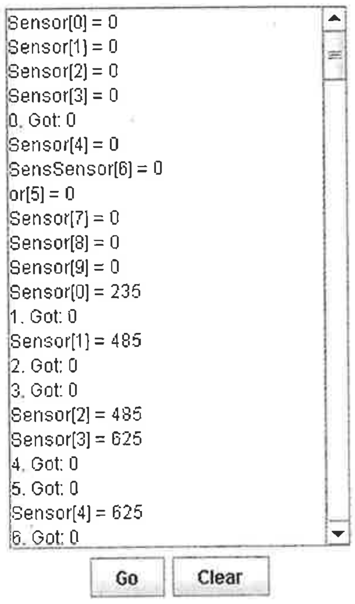
8.6 Animations

A popular task for a thread in Java is to control an animation. A thread process can direct the drawing of each frame while other aspects of the interface, such as responding to user input, can continue in parallel.

The `Drop2DApplet` program below illustrates a simple simulation of a bouncing ball using Java 2D drawing tools. The applet creates a thread to direct the drawing of the frames of the animation as the ball falls and bounces on the

floor and g
subclass of
the ball. Dr
The app
a thread to
first does sc
animation.
method in t
frame. If th
method, th

```
import
import
import

/** Thi
  * to
public

{
  // Wi
```



```
Sensor[0] = 0
Sensor[1] = 0
Sensor[2] = 0
Sensor[3] = 0
0. Got: 0
Sensor[4] = 0
SensSensor[6] = 0
or[5] = 0
Sensor[7] = 0
Sensor[8] = 0
Sensor[9] = 0
Sensor[0] = 235
1. Got: 0
Sensor[1] = 485
2. Got: 0
3. Got: 0
Sensor[2] = 485
Sensor[3] = 625
4. Got: 0
5. Got: 0
Sensor[4] = 625
6. Got: 0
```

Figure 8.7 Output of the `DataSyncApplet` program with `Sensor` and `DataGetter` classes.

floor and gradually comes to a rest (see Figure 8.8). The interface consists of a subclass of `JPanel` called `Drop2DPanel` and a button to initiate a new drop of the ball. `Drop2DPanel` displays the ball and calculates its position.

The applet implements `Runnable` and in the `start()` method it creates a thread to which it passes a reference to itself. The applet's `run()` method first does some initialization and then enters a loop that draws each frame of the animation. The loop begins with a 25 millisecond pause using the static `sleep()` method in the `Thread` class. Then the `Drop2DPanel` is told to paint the next frame. If the drop is done, the process jumps from the loop and exits the `run()` method, thus killing this thread.

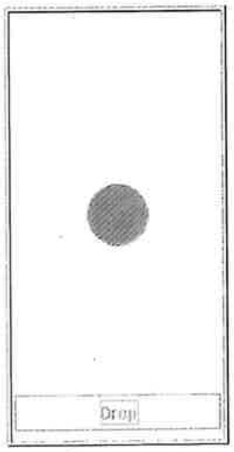


Figure 8.8 The `Drop2DApplet` program demonstrates how to create a simple animation by simulating a dropped ball that bounces when it hits the floor and gradually comes to rest.

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

/** This applet implements Runnable and uses a thread
  * to create a simple dropped ball demonstration.**/
public class Drop2DApplet extends JApplet
                       implements Runnable, ActionListener
{
  // Will use thread reference as a flag
```

```
Thread fThread = null;
Drop2DPanel fDropPanel;
JButton fDropButton;

/** Build the interface. **/
public void init () {
  Container content_pane = getContentPane ();
  Content_pane.setLayout (new BorderLayout ());

  // Create an instance of DropPanel
  fDropPanel = new Drop2DPanel ();
  // Add the Drop2DPanel to the content pane.
  Content_pane.add (BorderLayout.CENTER, fDropPanel);
  // Create a button and add it
  fDropButton = new JButton ("Drop");
  fDropButton.addActionListener (this);
  Content_pane.add (BorderLayout.SOUTH, fDropButton);
} // init

/** Start when browser is loaded or button pushed. **/
public void start () {
  // If the thread reference not null then a
  // thread is already running. Otherwise, create
  // a thread and start it.
  if (fTthread == null) {
     fThread = new Thread (this);
     fThread.start();
  }
} // start

/** Applet's stop method used to stop thread. **/
public void stop () {
  // Setting thread to null will cause loop in
  // run() to finish and kill the thread.
  fThread = null;
} // stop

/** Button command. **/
public void actionPerformed (ActionEvent ae){
  if (fDropPanel.isDone ()) start ();
}

/** The thread loops to draw each frame of drop. **/
public void run () {
  // Disable button during drop
  fDropButton.setEnabled (false);
```

```
    //
    fDr

    //
    whi

      t
      :
      c
      ,
      i
      :
    }
    //
    fDr
  } //
} // cla
```

The Drop2
the ball for
reverses the
friction. Ev
is done.

```
import :
import :
import :
import :

/** Thi
public c
{
  // Pa
  doubl
  // Co
  doubl
  doubl
  doubl
  // st
  doubl
  // Fr
  doubl
  // Fi
  boole
```

```
    // Initialize the ball for the drop.
    fDropPanel.reset ();

    // Loop through animation frames
    while (fThread!= null) {
        // Sleep 25msecs between frames
      try{Thread.sleep (25);
      }
      catch (InterruptedException e) {}
      // Repaint drop panel for each new frame
      fDropPanel.repaint ();
      if (fDropPanel.isDone ()) fThread = null;
    }
    // Enable button for another drop
    fDropButton.setEnabled (true);
  } // actionPerformed
} // class DropApplet
```

The Drop2DPanel class is shown below. The panel calculates the position of the ball for each increment of time between the frames and redraws the ball. It reverses the ball when it hits the floor and also subtracts some speed to simulate friction. Eventually, the ball comes to a rest and sets a flag that the drop simulation is done.

```
import javax.swing.*;
import java.awt.*;
import java.text.*;
import java.util.*;

/** This JPanel subclass displays a falling ball. **/
public class Drop2DPanel extends JPanel
{
  // Parameters for the drop
  double fY = 0.0, fVy = 0.0;
  // Conversion factor from cm to drawing units
  double fYConvert = 0.0;
  double fXPixel= 0.0, fYPixel = 0.0,
  double fRadius = 0.0, fDiam = 0.0;
  // starting point for ball in cm
  double fY0 = 1000.0;
  // Frame dimensions.
  double fFrameHt, fFrameWd;
  // Flag for drop status
  boolean fDropDone = false;
```

```
Ellipse2D fBall;

/** Reset parameters for a new drop. **/
void reset () {
  fFrameHt = getHeight ();
  fFrameWd = getWidth ();

  fXPixel = getWidth ()/2;
  fY = fY0; fVy = 0.0;

  // Conversion factor from cm to pixels
  // Start the ball about 20% from the top.
  fYConvert = fFrameHt / (1.2 * fY0);

  // Choose a size for the ball relative
  // to height of drawing area.
  fRadius = (int) ((0.1 * fY0) * fYConvert);
  fDiam = 2 * fRadius;

  // Make the ball
  fBall = new Ellipse2D.Double(fXPixel-fRadius,
                               fYPixel-fRadius,
                               fDiam, fDiam);
  setBackground (Color.WHITE);
  fDropDone = false;
} // reset

/** Draw the ball at its current position. **/
public void paintComponent (Graphics g) {
  super.paintComponent (g);
  Graphics2D g2 = (Graphics2D)g;
  // Antialiasing for smooth surfaces.
  g2.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
       RenderingHints.VALUE_ANTIALIAS_ON);

  // Determine position after this time increement
  calcPosition ();
  // Move the ball.
  fBall.setFrame(fXPixel-fRadius, fYPixel-fRadius,
                 fDiam,fDiam);
  // Want a solid red ball.
  g.setColor (Color.RED);
  g2.fill(fBall);
  // Now draw the ball
  g2.draw (fBall);
```

```
  }// paint

  /** Calcu
  void calc
    // Inc
    double

    // Calc
    fY = fY
    fVy = f

    // Conv
    fYPixel

    // Reve
    if ((fY
      fV
        //
        fV
        //
        //
        if

      }
    }
  } // calc

  /** Provi
  public bo
    return
  }
} // class
```

8.7 Timers

A timer provide
timer can:

- signal the redra
- issue periodic r
- trigger a single

As we saw in the
own simple time
action for a give

```
  }// paintComponent

  /** Calculate the ball position in the next frame. **/
  void calcPosition () {
    // Increment by 25 millseconds per frame
    double dt = 0.025;

    // Calculate position and velocity at each step
    fY = fY + fVy * dt – 490.* dt * dt;
    fVy = fVy – 980.0 * dt;

    // Convert to the pixel coordinates
    fYPixel = fFrameHt – (int)(fY * fYConvert);

    // Reverse direction when ball hits bottom.
    if ((fYPixel + fRadius) >= (fFrameHt-1)) {
         fVy = Math.abs (fVy);
         // Subtract friction loss
         fVy – = 0.1 * fVy;
         // Stop when speed at bottom drops
         // below an arbitrary limit
         if (fVy < 15.0) {
             fDropDone = true;
         }
    }
  } // calcPosition

  /** Provide a flag on drop status. **/
  public boolean isDone () {
    return fDropDone;
  }
} // class Drop2DPanel
```

Timer Applets

LIASING,

ment

lius,

8.7 Timers

A timer provides for periodic updates and scheduling of tasks. For example, a timer can:

- signal the redrawing of frames for an animation
- issue periodic reminders as with a calendar application
- trigger a single task, e.g. an alarm, to occur at a particular time in the future

As we saw in the previous section, with the `Thread` class you could create your own simple timer using the `Thread.sleep (long millis)` method to delay action for a given amount of time. This approach, however, has some drawbacks.

For periodic events, if the duration of processing in between the sleep periods varies significantly, then the overall timing will vary with respect to a clock. Also, if you need several timer events, the program will require several threads and this will use up system resources.

Java provides two timer classes [5–7]:

- `javax.swing.Timer` came with the Swing packages and is useful for such tasks as prompting the updating of a progress bar
- `java.util.Timer` and its helper class `java.util.TimerTask` provide for general purpose timers with more features than the Swing timer

These timers can provide multiple timed events from a single thread and thus conserve resources. They also have useful methods such as `scheduleAtFixedRate(TimerTask task, long delay, long period)` in `java.util.Timer`. This method will set events to occur periodically at a fixed rate and ties them to the system clock. This is obviously useful for many applications such as a countdown timer and an alarm clock where you don't want the timing to drift relative to absolute time.

8.7.1 `java.util.Timer` and `TimerTask`

The `Timer` and `TimerTask` combo in `java.util` offers the most general purpose timing capabilities and includes a number of options. A `Timer` object holds a single thread and can control many `TimerTask` objects. The `TimerTask` abstract class implements the `Runnable` interface but it does not provide a concrete `run()` method. Instead you create a `TimerTask` subclass to provide the concrete `run()` method with the code to carry out the task of interest.

In the example below, we create a digital clock using a timer to redraw a time display every second. The clock display uses `DateFormatPanel`, which we describe in Chapter 10 when discussing the date classes. Whenever this panel is drawn it displays the current time. The applet adds an instance of this panel to its content pane and in the `start()` method creates an instance of `java.util.Timer`.

A subclass of `TimerTask` called `UpdateTask` overrides the `run()` method and simply tells the panel to redraw itself. `UpdateTask` is defined as an inner class here and has access to the clock panel reference. The timer schedules calls to the `UpdateTask` every 1000 milliseconds. Figure 8.9 shows the clock display.

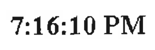

7:16:10 PM

Figure 8.9 The `ClockTimer1` and `ClockTimer2` programs, which both provide a current time display like that shown here, illustrate the use of `java.util.Timer` and `javax.swing.Timer`, respectively.

```
import javax.swing.*;
import java.awt.*;
import java.util.*;

/** This applet implements Runnable and uses a thread
```

```
  * to create a digital clock. **/
public class ClockTimer1 extends Japplet
{
  java.util.Timer fTimer;
  // Need panel reference in run().
  DateFormatPanel fClockPanel;

  public void init () {
    Container content_pane = getContentPane ();

    // Create an instance of DrawingPanel
    fClockPanel = new DateFormatPanel ();

    // Add the DrawingPanel to the contentPane.
    content_pane.add (fClockPanel);
  }

  public void start () {
    // Create a timer.
    fTimer = new java.util.Timer ();

    // Start the timer immediately and then repeat calls
    // to run in UpdateTask object every second.
    fTimer.schedule (new UpdateTask (), 0, 1000);
  }

  /** Stop clock when web page unloaded. **/
  public void stop () {
    // Stop the clock updates.
    fTimer.cancel ();
  }

  /** Use the inner class technique to define the
    * TimerTask subclass to update the clock.**/
  class UpdateTask extends java.util.TimerTask {
    public void run () {
      fClockPanel.repaint ();
    }
  }
} // class ClockTimer1
```

(Note that since we import both `javax.swing.*` and `java.util.*` it is necessary to use the fully qualified type `java.util.Timer` when declaring the `fTimer` variable. Without the full qualification, the compiler would not know whether we wanted `javax.swing.Timer` or `java.util.Timer`.)

8.7.2 `javax.swing.Timer`

Although it has fewer options, the `javax.swing.Timer` can do some of the same basic timing tasks as `java.util.Timer`. Below we show another version of the digital clock except that it uses `javax.swing.Timer`. This timer contacts an `ActionListener` after every time period rather than a `TimerTask` object. Here the applet implements the `ActionListener` interface. The constructor for the timer takes as arguments the update period value and the reference to the applet. The timer is then started and after every second the `actionPerformed()` method will be invoked and the clock panel repainted. The applet's `stop()` method stops the timer.

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;

/** This applet implements Runnable and uses a thread
  * to create a digital clock. **/
public class ClockTimer2 extends JApplet
        implements ActionListener
{
  javax.swing.Timer fTimer;

  // Need panel reference in run().
  DateFormatPanel fClockPanel;

  public void init () {
    Container content_pane = getContentPane ();

    // Create an instance of DrawingPanel
    fClockPanel = new DateFormatPanel ();

    // Add the DrawingPanel to the contentPane.
    content_pane.add (fClockPanel);
  }

  public void start () {
    // Send events very 1000ms.
    fTimer = new javax.swing.Timer (1000, this);

    // Then start the timer.
    fTimer.start ();
  }
```

```
    // Timer creates an action event.
    public void actionPerformed (ActionEvent e) {
      Object source = e.getSource ();
      if (source == fTimer)
        fClockPanel.repaint ();
    }

    // Stop clock when web page unloaded
    public void stop () {
      // Stop the clock updates.
      fTimer.stop ();
    }
  } // class ClockTimer2
```

8.8 Concurrency utilities in J2SE 5.0

Java Release 5.0 adds numerous enhancements to the threading control and concurrency features of Java. Some of the enhancements are advanced features beyond the scope of this book, and others require an understanding of the new generics feature of 5.0. So we defer discussion of these until after we have explained generics in Chapter 10.

8.9 Web Course materials

The Web Course Chapter 8: *Supplements* section provides additional information and examples dealing with threading. This includes additional discussion of the new `java.util.concurrent` tools available with Java 5.0.

In the Chapter 8: *Tech* section we expand the number of histogram classes and subclasses as we add new capabilities. For example, we create an adaptive histogram class that can expand its range limits as new data arrives. We use timers to simulate the reading of data to plot in a histogram. We also discuss sorting tools in Java and use them to sort the bins in a histogram according to the number of entries in the bins. We use a thread to animate the sorting of a histogram.

This increase in histogram classes illustrates a common challenge in object-oriented programming: when to modify existing classes, when to create subclasses, and when to create whole new classes. Subclasses would seem the logical answer for an OOP environment but many small revisions for every new option that comes along can quickly lead to an unmanageable plethora of subclasses. Eventually, your entire class design may need to be re-worked (also called *refactoring*, with the implication that common parts are *factored out* into a common superclass). We discuss class design and refactoring further in the *Tech* section. The *Physics* section looks at issues involved in animating simulations.

References

[1] Scott Oaks, Henry Wong, *Java Threads,* 2nd edn, O'Reilly, 1999.
[2] *Lesson: Threads: Doing Two or More Tasks At Once – The Java Tutorial*, Sun Microsystems, `http://java.sun.com/docs/books/tutorial/essential/threads/`.
[3] *How to Use Threads in Creating a GUI with JFC/Swing – The Java Tutorial*, Sun Microsystems, `http://java.sun.com/docs/books/tutorial/uiswing/misc/threads.html`.
[4] *AWT Threading Issues – Java 2 Platform, Standard Edition, API Specification*, `http://java.sun.com/j2se/1.5/docs/api/`.
[5] *Using the Timer and TimerTask Classes – The Java Tutorial*, Microsystems, `http://java.sun.com/docs/books/tutorial/uiswing/misc/timer.html`.
[6] *Using Timers to Run Recurring or Future Tasks on a Background Thread*, JDC Tech Tips, May 30, 2000, `http://java.sun.com/developer/TechTips/2000/tt0530.html#tip2`.
[7] John Zukowski, *Using Swing.Timers*, JDC Tech Tips, May 21, 2002, `http://java.sun.com/developer/JDCTechTips/2002/tt0521.html`.

Chapte
Java i

9.1 Intro

Java provide
on the conc
one directio
correspondi
output strea
into a progr
a network p
in data thro
The bulk
in Figure 9.
`OutputSt`
output tasks
add more ca
as a file or
filtering the
Package

- `java.io`
- `java.ni`
concept o
other entit
capabilitie
- `java.ne`
- `java.ut`
- `java.ut`
- `javax.i`
I/O, inclu
Chapter 1

Java I/O is
[1,2]. Here

JavaTech is a practical introduction to the Java programming language with an emphasis on the features that benefit technical computing, such as platform independence, extensive graphics capabilities, multi-threading, and tools to develop network and distributed computing software and embedded processor applications.

The book is divided into three parts. The first part presents the basics of object-oriented programming in Java and then examines topics such as graphical interfaces, thread processes, I/O, and image processing. The second part begins with a review of network programming and develops Web client-server examples for tasks such as monitoring of remote devices. The focus then shifts to distributed computing with RMI, which allows programs on different platforms to exchange objects and call each other's methods. CORBA is also discussed and a survey of Web services is presented. The final part examines how Java programs can access the local platform and interact with hardware. Topics include combining native code with Java, communication via serial lines, and programming embedded processors.

JavaTech demonstrates the ease with which Java can be used to create powerful network applications and distributed computing applications. It can be used as a textbook for introductory- or intermediate-level programming courses, and for more advanced students and researchers who need to learn Java for a particular task.

The book is supported by an extensive website with instructional materials, applets and application codes:

www.cambridge.org/9780521821131
- applets and application codes
- exercises
- extensive references
- resources and tips
- starter and demo programs
- supplementary material for advanced users
- updated regularly

CLARK S. LINDSEY runs his own company, which develops Java applications, Web publications, and educational tools and materials.

JOHNNY S. TOLLIVER develops robust Web services software and GPS applications at Oak Ridge National Laboratory. He is a Sun Certified Java Programmer.

THOMAS LINDBLAD is a professor in the Department of Physics at the Royal Institute of Technology, Stockholm, where he researches techniques in data analysis in high data rate systems.

Cover illustration: a space radar image of the summits of two large volcanoes, Mt. Merbabu (mid center) and Mt. Merapi (lower center), in Central Java, Indonesia. Lava flows of different ages and surface roughness appear in shades of green and yellow surrounding the summit. This image was taken on October 10, 1994 by the Spacebourne Imaging Radar-C/X-Band Synthetic Aperture Radar on the space shuttle Endeavour. Courtesy of NASA JPL / Visible Earth

Cover designed by Hart McLeod

CAMBRIDGE
UNIVERSITY PRESS
www.cambridge.org

ISBN 0-521-82113-4

9 780521 821131

APPENDIX C

Internet Technology as a Tool for Solving Engineering Problems

Aleksander Malinowski
Department of Electrical and Computer Engineering
Bradley University, Peoria, Illinois 61625, USA
olekmali@ieee.org, http://cegt201.bradley.edu/~olekmali/

Bogdan Wilamowski
College of Engineering at Boise
University of Idaho, Boise, Idaho 83712, USA
wilam@ieee.org, http://nn.uidaho.edu/

Abstract - **This tutorial covers all primary technologies that can be used for Web programming with applications to the Internet based data acquisition and system control. The presentation is divided into two parts. The first part discusses several programming languages as programming tools. It provides in depth discussion of the tasks that are best to implement with and on the advantages versus challenges associated with particular cases. The following tools are covered: HTML, JavaScript, Java Applets, Cookies, CGI, PERL, PHP, native languages (C/C++), and Web server configuration for security. Special emphasis is given to PERL and JAVA. Several programming examples for client, server and client-server applications are provided. The fragments of code are selected so that they provide good jumpstart to programming in particular languages for anybody with good programming skills in any programming language (preferably C++ or C). Advantages and disadvantages of different computer languages are discussed so a proper programming platform for different applications and task can be chosen.**

I. SYSTEM ARCHITECTURE CONSIDERATIONS

During software development, it is important to justify which part of the software should run on the client machine and which part should run on the server. Sometimes even the very fundamental client-server architecture must be reconsidered in favor of a peer to peer decentralized structures. The decision about the architecture can be made either based on the process control strategy or based on the information storage.

In case of the process control approach, the first approach is used when there each of the controlled objects can considered to be separate from possible other similar objects. The latter architecture is more beneficial in case of many controlled objects that cooperate with each other. When the information storage is considered then client server is favored over peer to peer communication in cases where information must be centralized, or is easier to manage when it is centralized.

Even between these two models, there may be a hybrid. Consider an instance when one controls a process that is implemented by many objects that cooperate with each other. The controller either deals with each object separately using a client-server approach, or deals only with one object and then relies on the peer to peer architecture to carry out the request. The latter case adds additional complexity of dealing with a distributed server.

II. COMPONENT PARTITIONING AND DATA FLOW

Once a particular architecture is chosen, the component partitioning needs to be considered. Peer to peer architecture usually yields symmetry of all objects. The decisions are made for client-server based on several factors:

- Amount of memory and CPU power available for server and clients. These restrictions may be imposed by technological or cost restrains.
- Available bandwidth of the network connection.
- Connection reliability and latency, especially in case of closing the control loop via network.
- Ease of installation or no need to preinstall any specific component on a client machine.

It is possible to develop two dedicated software components, one for server, and another one for a client and preinstall both. However, other strategies allow for more flexibility such as an automatic installation or update of the client side-software from the server. The latter approach requires storage of the client software components on the server object, possibly increasing the memory requirements and the initial network traffic when a new client must to be installed or updated.

Regardless from the choice of just in time downloaded or preinstalled client the software designer must make choices regarding partitioning the tasks between the server and the client. In case of control, the best results are achieved when the control loop is closed locally on the server that is installed on the controlled. The Internet bandwidth is already adequate for many applications if their data flow is carefully designed. Furthermore, the bandwidth limitation will significantly improve with time. It is therefore important to develop methods, which take advantage of networks and then platform independent browsers. This would require solving several issues, such as:

- Minimization of the amount of data which must be sent by a network
- Task partitioning between the server and client
- Selection of programming tools used for various tasks
- Development of special user interfaces
- Use of multiple servers and job sharing among them
- Security, privacy and, in case of pay per use, account handling
- Portability of software used on servers and clients

- Distributing and installing network packages on several servers

Fig. 1 illustrates an example of software component partitioning for a semiautonomous remote controlled robot. This particular application utilizes several client-server partitions for multiple components. In addition, the network server is at the same time the client in the relation to the thin embedded server that controls the robot movements on the lowest software level.

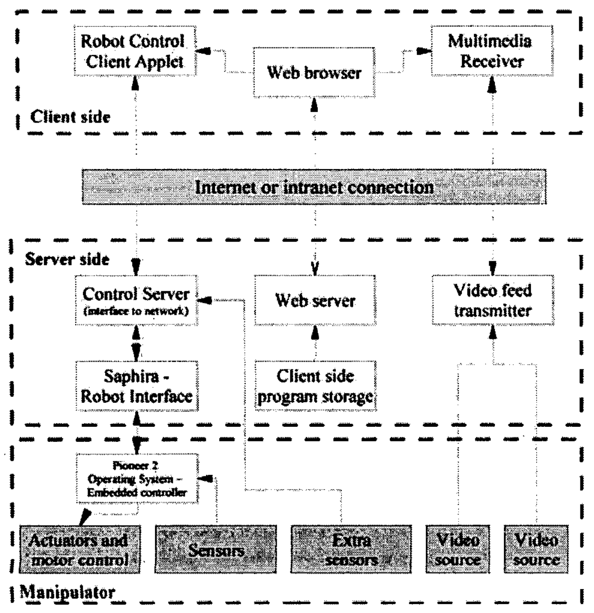

Robot Control Client Applet
Web browser
Multimedia Receiver
Client side
Internet or intranet connection
Server side
Control Server (interface to network)
Web server
Video feed transmitter
Saphira - Robot Interface
Client side program storage
Pioneer 2 Operating System - Embedded controller
Actuators and motor control
Sensors
Extra sensors
Video source
Video source
Manipulator

Figure 1. Example of client –server component partition.

Choosing the right set of software tools to implement the components of the system is the next dilemma to be solved after the decisions about the data flow among the software components that are distributed in the network are made. This problem is addressed in the next section.

III. MOST COMMONLY USED NETWORK PROGRAMMING TOOLS

Although it is possible to develop network applications using solely C++, or other compiled languages, it is much easier to develop networked applications using dedicated software tool for each component. There are several well-developed network-programming tools available today [1]. These tools include HTML, JavaScript, VBScript, Java, ActiveX, Common Gateway Interface (CGI) and PERL or C++, Active Server pages (ASP) and PHP. It is essential to make a correct decision which programming language should be used for which part of the software package. Short characterizations of different network programming tools are given below.

A. Hypertext Markup Language

Hypertext Markup Language (HTML) was originally designed to describe a document layout regardless of the displaying device, its size, and other properties [2]. It can be incorporated into networked application front-end development either to create form-based dialog boxes or as a tool for defining the layout of an interface, or wraparound for Java applets or ActiveX components. In a way, HTML can be classified as a programming language because the document is displayed as a result of the execution of its code. In addition scripting language can be used to define simple interactions between a user and HTML components [3][4]. Several improvements to the standard language are available: Cascading style sheets (CSS) allow very precise description of the graphical view of the user interface; Compressed HTML allows bandwidth conservation but can only be used by Microsoft Internet Explorer. HTML is also used directly as it was originally intended – as a publishing tool for instruction and help files that are boundled with the software.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2//EN">
<HTML>
<HEAD>
<TITLE>This is the title for this Web
Page</TITLE>
<META HTTP-EQUIV="Content-Type"
CONTENT="text/html; charset=iso-8859-1">
<META HTTP-EQUIV="Pragma" CONTENT="no-cache">
<META NAME="ROBOTS" CONTENT="INDEX,NOFOLLOW">
<META NAME="Author" CONTENT="BMW & AM">
<META NAME="Description" CONTENT="This is
displayed by the search engine">
<META NAME="KeyWords"
CONTENT="search engine kewoards, html">
</HEAD>
<BODY BGCOLOR="white" TEXT="black">
<H1>This is the title</H1>
<P ALIGN="left">This is the body of
this Web page.
<A HREF="http://nn.uidaho.edu">go</A>
<P ALIGN="center">Another paragraph and
<FONT COLOR="red" FACE="Ariel,helvetica"
SIZE="+1">a different font</FONT>
<!-- this is a coment -->
</BODY>
</HTML>
```

Figure 2. Typical HTML source code.

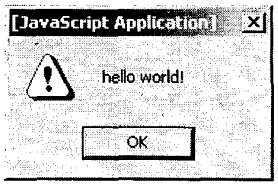
The HTML code shown in Fig.2 illustrates the nature of this language. The control structures are called tags. A tag is identified by < and > and us used to control the meaning and format that is used to display the information. Most of the tags are used in pairs, for example <BODY> and </BODY> marks the beginning and the end of the section that should be displayed as a Web page. Each tag may have several attributes. For example the two of many attributes of `<BODY ...>` are color setting `BGCOLOR` and `TEXT`. Inside the body of the page tags are used to provide text formatting. <P...> denotes a new paragraph, and is one of only a few tags that do not need the complementing and end tag `</P>`. `<Hn>` indicates the n-th header or section title of the n-th level. In addition to those and many other logical information tags, there are several tags that porvide only instruction regarding the way the text is to be displayed, for example `<FONT...>` tag. Although it is possible to set a particular font size in points, it is strongly recommended to alter the readers preference using relative sizes like +1 in the example above. The reader should be able to adjust the display to her preferences so that it is easy to read.

The anchor tag `<A ...>` is the most important feature of the HTML. This implements the very idea of hypertext – the links. The `HREF` attribute instructs the Web browser about the location of another page that must be loaded in case the reader clicks on the text enclosed until `</A>`.

The header portion of the Web page that is marked by `<HEAD>` and `</HEAD>` may seam not to be that important. Information enclosed there may be very important for Web browsers, proxy systems or search engines. The example in Fig. 2 instructs the Web browser always to check for the new version of the Web page (`no-cache`), and defines the font set (`8859-1`) that is very important when the Web page displays any non-English characters. The other tags (robots, author, keywords and description) are sued by search engines to enhance the automatic classification of the Web page.

B. JavaScript

HTML itself lacks even basic programming constructs such as conditional statements or loops. A few scripting interpretive languages were developed to allow for use of programming in HTML [2]. They can be classified as extensions of HTML and are used to manipulate or dynamically create portions of HTML code. One of the most popular among them is JavaScript. The only drawback is that although JavaScript is already well developed still there is no one uniform standard. Different Web browsers may vary a little in the available functions [4]. JavaScript is an interpretative language and the scripts are run as the Web page is downloaded and displayed. There is no strong data typing or function prototyping. Yet the language includes support for object oriented programming with dynamically changing member functions. JavaScript programs can also communicate with Java applets that are embedded into an HTML page.

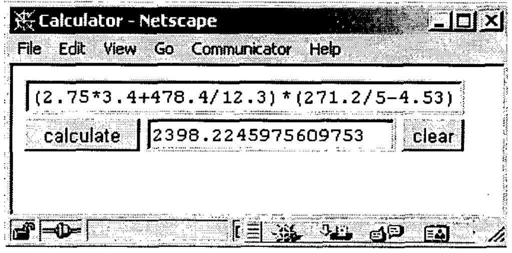


```
<SCRIPT language="JavaScript">
// this comment goes to the end of the line
alert("hello world!");
// end hiding comment
</SCRIPT>
<NOSCRIPT>No script support found</NOSCRIPT>
```

Figure 3. A simple example of JavaScript code.

JavaScript is part of the HTML code. It can be placed in both header and body of a Web page. The script starts with `<script language="JavaScript">` line. This example generates an alert dialog box shown above the code.

One of the most useful applications of JavaScript is verification of the filled form before it is submitted on-line. That allows for immediate feedback and preserves the Internet bandwidth as well as lowers the Web server load. Fig. 4 shows a sample code of an HTML form and its interaction with JavaScript that responds immediately.



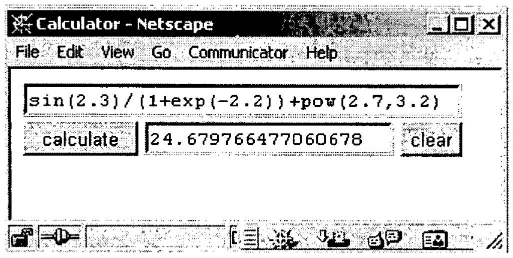
```
(2.75*3.4+478.4/12.3)*(271.2/5-4.53)
2398.2245975609753
```

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0//EN">
<html>
<head>
<TITLE>Calculator</TITLE>
</head>
<body>
<form name="form1">
<input type="text" name="text1" size="36"><BR>
<input type="button" name="button1"
    value="calculate"
    onclick=document.form1.text2.value
        =eval(document.form1.text1.value)>
<input type="text" name="text2">
<input type="reset" value="clear">
</form>
</body>
</html>
```

Figure 4. A Web page with JavaScript based calculator.

The next example shows a more powerful calculator, which is capable to compute even complicated functions.

Note that all computations are done not on the server but on the client computer. The web page generated is similar to this shown in the previous example but it is much more powerful. Its view and source code is shown in Fig. 5.



```
sin(2.3)/(1+exp(-2.2))+pow(2.7,3.2)
24.679766477060678
```

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0//EN">
<html>
<head>
<TITLE>Calculator</TITLE>
</head>
<body>
<script language="JavaScript">
function math() {
var s=document.form1.text1.value;
s1=s.replace("abs", "Math.abs");
s2=s1.replace("sin", "Math.sin");
s3=s2.replace("cos", "Math.cos");
s4=s3.replace("sqrt", "Math.sqrt");
s5=s4.replace("tan", "Math.tan");
s6=s5.replace("atan", "Math.atan");
s7=s6.replace("asin", "Math.asin");
s8=s7.replace("acos", "Math.acos");
s9=s8.replace("exp", "Math.exp");
s10=s9.replace("floor", "Math.floor");
s11=s10.replace("log", "Math.log");
s12=s11.replace("max", "Math.max");
s13=s12.replace("min", "Math.min");
s14=s13.replace("pow", "Math.pow");
s15=s14.replace("random", "Math.random");
s16=s15.replace("round", "Math.round");
document.form1.text2.value=eval(s16);
}
</script>
<form name="form1">
<input type="text" name="text1" size=36>
<input type="button" name="button1"
value="calculate" onclick="math()">
<input type="text" name="text2"><BR>
<input type="reset" value="clear">
</form>
</body>
</html>
```

Figure 5. A Web page with advanced calculator.

C. Visual Basic Script

If the client-side software development is limited to Microsoft Windows and Microsoft Internet Explorer then VBScript may be used instead of JavaScript. The disadvantage is the lack of portability that is offered by this tool. However, that downside is compensated by ease of communicating with ActiveX components and possibility to use programs and libraries available to the operating system [2].

D. Java

Java is an object oriented programming language compiled in two stages. The first stage of compilation, to so-called byte-code, is performed during the code development. Byte-code can be compared to machine code instructions for a microprocessor [5]. Because no processor understands directly byte-code instructions, interpreters, called Java Virtual Machines (JVM) were developed for various microprocessors and operating systems. At some point JVM were improved so that instead of interpreting the code they do perform the second stage of compilation, directly to the machine language. However, to cut down the initial time to run the program the compilation is done only as necessary (just in time (JIT)) and there is no time for extensive code optimization [6]. At current state of art of JIT technology, programs written in Java run about two to five times slower than their C++ counterparts. Adding a JVM to a Web browser allowed embedding software components that could be run on different platforms [5][7].

Several features ensured success and increasing importance of this programming tool:

- similarity to C and C++ - a lot of existing programmers can switch relatively easily [5][7][8][9]
- support of C++ objects – suitability for large projects [5][8][9]
- simplified features – less complex than C++, easier to learn and utilize correctly [7][8][9]
- large standard set of libraries, including graphical libraries that can be used on multiple OS platforms [7][9]
- built in network libraries and some IP protocols [7][9][10]
- simple, platform independent multithreading – not as powerful as in C or C++ but much simpler [7][9]
- availability of fast JVM that use JIT compiler technology – only two times slower than C++ [6][7]
- ability to control the level of security buy enabling or disabling certain libraries that come with JVM
- availability of non-portable features by linking functions in machine language of a particular system [9]
- Smaller requirements for flash memory in the embedded systems due to compactness of byte-code (but more volatile memory is required)

Despite all those great advantages, there are a few problems of implementation that prevents Java from being used everywhere.

- it is still at least two to five times slower than C++ [6]
- multithreading does not have all features available to C or C++ programs [7][8]
- Most of implementations of JVM do not allow real time running due to garbage collector type of the memory management [6]
- Much higher memory requirements for JVM and running program (but smaller footprint of applications stored in flash memory versus C/C++)

D. ActiveX

Microsoft developed ActiveX as another technology allowing for the automatic transfer of software over the network [2][11]. ActiveX, however, can be executed presently only on a PC with a Windows operating system, thus making the application platform dependent. Although this technology is very popular already, it does not allow for the development of applications running on multiple platforms. ActiveX components can be developed in Microsoft Visual Basic or Microsoft Visual C++. There are the only choice in cases when Java is too slow, or when some access to the operating system functionality or devices supported only by Windows OS is necessary. The easy access to the operating system form an ActiveX component makes it impossible to provide additional security by limiting the features or resources available to the components.

Fig. 6 shows one of the simplest possible programs written in Java that also demonstrates use of functions. Since the language is strongly object oriented, all functionsmust be embedded in a class.

```
public class Test {
    public static void main(String args[]) {
        // a comment
        procedure("Hello programmer!");
    }
    private static void procedure(String s) {
        System.out.println(s);
    }
}
```

Figure 6. Code for a simple application written in Java.

Fig. 7 and Fig. 8 show a template for an applet written in Java. Applets are run embedded in Web pages. Fig. 7 shows how to embed the applet in HTML.

```
<APPLET CODEBASE="." CODE="Test.class"
    WIDTH="200" HEIGHT="100">
</APPLET>
```

Figure 7. Embedding an applet in a Web page.

Function `paint()` is called from the operating system environment whenever the graphics needs to be redrawn. Functions `init()` and `start()` are called when the applet is initialized. All computations should be initialized there and then carried on in a separate thread. Function `stop()` is called when the applet need to be stopped. All computations that were initialized in `start()` and carried on in another threads must be stopped then. This simple applet does not do anything besides painting a text and drawing two horizontal lines.

```
// a sample applet template
import java.applet.Applet;
import java.awt.Graphics;
public class Test extends Applet {
    public void init() {
    }
    public void start() {
    }
    public void paint(Graphics g) {
      g.drawLine(10,30,  120, 30);
      g.drawLine(10,60,  120, 60);
      g.drawString("Hello Programmer!", 10, 50);
    }
    public void stop() {
    }
}
```

Figure 8. A template for an applet written in Java.

E. CORBA and DCOM

CORBA (Common Object Request Broker Architecture) is a technology developed in the early 90's for network distributed applications. It is a protocol for handling distributed data, which has to be exchanged among multiple platforms [12][13]. A CORBA server or servers must be installed to access distributed data. CORBA in a way can be considered as a very high-level application programming interface (API). It allows sending data over the network, sharing local data that are registered with the CORBA server among multiple programs. Microsoft developed its own proprietary API that works only in Windows operating system. It is called DCOM and can be used only in ActiveX technology [11][14].

F. Common Gateway Interface

CGI, which stands for Common Gateway Interface, can be used for the dynamic creation of web pages. Such dynamically created pages are an excellent interface between a user and an application run on the server [2][9] [15]. CGI program is executed when a form embedded in HTML is submitted or when a program is referred directly via a Web page link. The Web server that receives a request is capable of distinguishing whether it should return a Web page that is already provided on the hard drive or run a program that creates one. Any such program can be called a CGI script. CGI describes a variety of programming tools and strategies. All data processing can be done by one program, or one or more other programs can be called from a CGI script. The name CGI script doe not denote that a scripting language must be used. However, developers in fact prefer scripting languages, and PERL is the most popular one.

Because of the nature of the protocol that allows for transfer of Web pages and execution of CGI scripts there is a unique challenge that must be faced by a software developer. Although users working with CGI-based programs have the same expectations as in case of local user interface, the interface must be designed internally in entirely different way. The Web transfer is a stateless process. That means, that no information is sent by Web browsers to the Web servers that identify each user. Each time the new user interface is sent as a Web page, it must contain all information about the current state of the program. That state is recreated each time a new CGI script is sent and increases the network traffic and time latency caused by limited bandwidth and time necessary to process data once again.

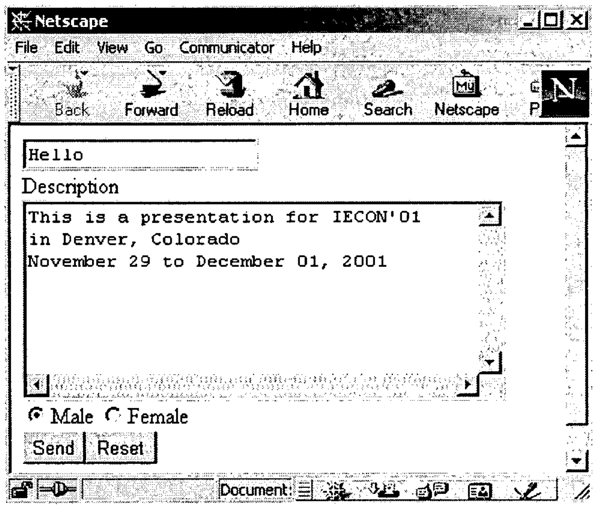
In addition, the server side software must be prepared for inconsistent data streams. For example, a user can back off through one or more Web pages give a different response to a particular dialog box and execute the same CGI script. At the time of the second execution of the same script, the data sent back with the request may already be out of synchronization from the data kept on server. Therefore, additional validation mechanisms must be implemented in the software that are not necessary in case of a single program.

G. PERL

PERL is an interpretive language dedicated for text processing. It is primarily used as a very advanced scripting language for batch programming and for text data processing [2][16][17]. PERL interpreters have been developed for most of existing computer platforms and operating systems. Modern PERL interpreters are in fact not interpreters but compilers that precompile the whole script before running it.

PERL was originally developed for Unix as a scripting language that would allow for automation of administrative tasks. It has many very efficient string, data stream and file processing functions. Those functions make it especially attractive for CGI processing that deals with reading data from the networked streams, executing external programs, organizing data, and in the end producing the feedback to the user in the form of a text based HTML document that is sent back as an update of the user interface [2][15]. Support of almost any possible computing platform and OS and existence of many program libraries makes it a platform independent tool.

Fig. 9 and Fig. 10 show an example of a data form that is filled in by a user on a remote computer (client). After the form shown in Fig. 9 is completed, the user clicks the "SEND" `submit` button. All data is transferred to the server and forwarded to the CGI script that is specified in the `form` tag in the `action` attribute. The source code of the CGI program is shown in Fig. 10. The program reads the data, processes them, and generates a Web page that is a feedback to the user.


Netscape
File Edit View Go Communicator Help
Back Forward Reload Home Search Netscape
Send Reset
Document:

Hello

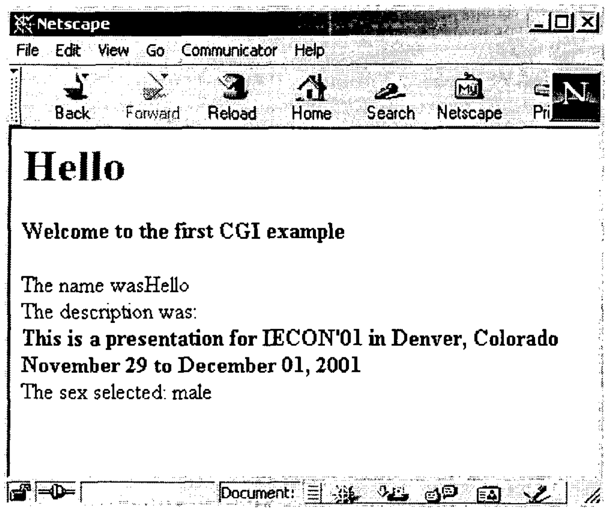
Description

```
This is a presentation for IECON'01
in Denver, Colorado
November 29 to December 01, 2001
```

[✓ Male] Female

```
<FORM action="http://nn.uidaho.edu/csp/cgi1.pl"
method="get">
<INPUT TYPE="text" name="name"><BR>
Description<BR>
<TEXTAREA name="description" rows=5 cols=40>
</TEXTAREA><BR>
<INPUT type="radio" name="sex" value="male">Male
<INPUT type="radio" name="sex"
value="female">Female
<BR>
<INPUT type="submit" value="Send"><INPUT
type="reset">
</FORM>
```

Figure 9. Data form implemented in HTML.


Netscape
File Edit View Go Communicator Help
Back Forward Reload Home Search Netscape Pri
Document:

Hello

Welcome to the first CGI example

The name wasHello
The description was:
This is a presentation for IECON'01 in Denver, Colorado November 29 to December 01, 2001
The sex selected: male

```
#!c:\progra~1\perl\bin\perl.exe
use strict;
use CGI qw(:standard);
print header;
print <<label1;
<H1><FONT COLOR="#FF0000"> Hello </FONT></H1>
<B>Welcome to the first CGI example </B><P>
label1
print "The name was", param('name'), "<BR>";
print "The description was:<B><BR>",
param('description'),"<BR></B>";
print "The sex selected: ",
param('sex'), "<P>";
```

Figure 10. A CGI-script written in PERL that handles data received from the form shown in Fig. 9.

The PERL code above uses CGI library with `param` function and this way reading data from the form is very simple. For example `param('name')` returns a string that was typed in the text field named `name` (see the HTML code above). `param('sex')` returns the name of the radio button pressed. `param('description')` returns a string that was typed in the text area named `description`. The PERL code generates a new screen on the client computer as shown above the code.

Please note that there are two ways of displaying messages of the client computer. The first

```
print <<label1;
<H1><FONT COLOR="#FF0000"> Hello </FONT></H1>
<B>Welcome to the first CGI example </B><P>
label1
```

sends entire HTML code between lines `print <<label1;` and `label1`. The other way is to use print statement and send HTML code line by line using `print` statements.

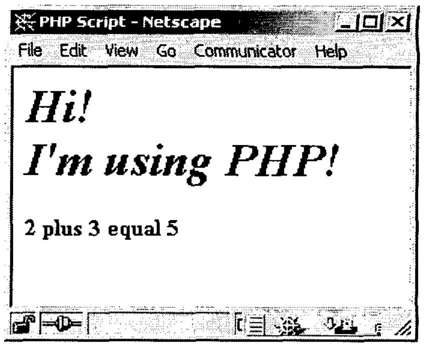
H. Active Server Pages

The concept of CGI scripts is centered on the idea that a program that is external to the Web server is run on the request made by a client. Then an HTML based reply is generated and sent back as the part of the outcome of the execution. Active Server Pages (ASP) provide the same functionality with the exception that the external program or programs are embedded into the skeletons of Web pages [**18**]. Those pages are preprocessed by the Web server before they are forwarded to the client, and the outcome of the embedded scripts is included.

In case of a CGI script, a reply to the user by sending an HTML based Web page is its significant portion. It makes sense then to provide also tools for embedding the scripts inside HTML instead of embedding HTML inside print statements in the CGI script. ASP technology is nothing else but shifting the way the server side programs are organized.

I. PHP

PHP is a scripting language like PERL. In fact, its syntax resembles PERL. The main difference lays in the set of standard built in libraries that support generation of HTML code, processing data from and to the Web server, and handling cookies. The same functionality can be accessed in PERL by inclusion of one or more libraries. PHP can be sued either as a classical CGI scripting language or as an implementation of ASP technology [**18**]. Since certain frequently used functionality is built in directly into the language, it is more efficient to use. In general any specialized tool will be somewhat more efficient for one particular task it was designed for, instead of other powerful but general purpose tools. PHP has been very popular for the last three years.



Hi!
I'm using PHP!

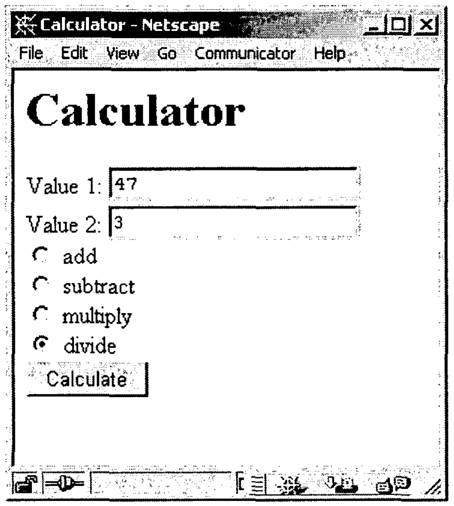
2 plus 3 equal 5

```
<HTML>
<HEAD><TITLE>PHP Script</TITLE></HEAD>
<BODY>
<?
echo "<H1><em>Hi! <BR>
I'm using PHP!</em></H1>";
$a = 2; $b = 3; $c=$a+$b;
echo "<B>$a plus $b equal $c </B></p>";
?>
</BODY>
</HTML>
```

Figure 11. A simple server side script in PHP.

PHP script (between `<?` and `?>`) can be easily incorporated into HTML code as illustrated in Fig. 11. Instead of `<?` and `?>` one can use `<?php` and `?>`, or `<script language="php">` and `</script>`. The script is run by the Web server on the server side, before the Web page is transferred through the Internet to the browser.

Fig. 12 shows another example of PHP programming. This time PHP is used to generate an HTML form. It is more concise than HTML and thus faster to develop and less likely to contain errors, but add to the load of the server computer. The resulting form is shown above the source code.



Calculator

Value 1: 47

Value 2: 3

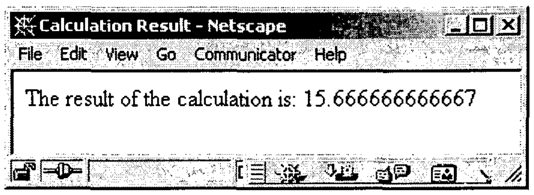
- ( ) add
- ( ) subtract
- ( ) multiply
- (•) divide

Calculate

```
<HTML>
<HEAD><TITLE>Calculator</TITLE></HEAD>
<BODY>
<h1>Calculator</h1>
<FORM METHOD="post" ACTION="calculator.php">
Value 1: <INPUT TYPE="text" NAME="val1"></br>
Value 2: <INPUT TYPE="text" NAME="val2"></br>
<INPUT TYPE="radio" NAME="calc" VALUE="add">
add<br>
<INPUT TYPE="radio" NAME="calc" VALUE="sub">
subtract<br>
<INPUT TYPE="radio" NAME="calc" VALUE="mul">
multiply<br>
<INPUT TYPE="radio" NAME="calc" VALUE="div">
divide</br>
<INPUT TYPE="submit" NAME="submit"
VALUE="Calculate">
</FORM>
</BODY>
</HTML>
```

Figure 12. PHP utilized to generate a form in HTML.

When the form is submitted, the Web server needs to run a CGI script. Since a Web page merged with PHP can function as a program the PHP-based page can be used for the form processing as shown in Fig. 13. The Web page that is generated in the reply is shown above the source code.


Calculation Result - Netscape
File Edit View Go Communicator Help

The result of the calculation is: 15.666666666667

```
<?
if (($val1 == "") || ($val2 == "") || ($calc
=="")) {
        header("Location:
http://nn2/cal_fm.htm");
        exit;
}
if ($calc == "add") {$r = $val1 + $val2;}
else if ($calc == "sub") {$r= $val1-$val2;}
else if ($calc == "mul") {$r = $val1*$val2;}
else if ($calc == "div") {$r = $val1/$val2;}
?>
<HTML><HEAD>
<TITLE>Calculation Result</TITLE> </HEAD>
<BODY>
The result of the calculation is: <? echo "$r";
?>
</BODY>
</HTML>
```

Figure 13. PHP utilized for Cgi scripting.

One of the principles of the correct coding is enclosing all source code that implements a particular functionality in one place. This can be applied to PHP. The code shown in Fig. 14 works both as HTML form generator and as the data processor in case it is called back by the generated form.

```
<HTML> <HEAD> <TITLE>AIO Form</TITLE> </HEAD>
<BODY>
<?
$formstring = "
<FORM METHOD=\"post\" ACTION=\"$PHP_SELF\">
Value 1: <INPUT TYPE=\"text\" NAME=\"val1\"></br>
Value 2: <INPUT TYPE=\"text\" NAME=\"val2\"></br>
<INPUT TYPE=\"radio\" NAME=\"calc\"
VALUE=\"add\"> add<br>
<INPUT TYPE=\"radio\" NAME=\"calc\"
VALUE=\"sub\"> subtract<br>
<INPUT TYPE=\"radio\" NAME=\"calc\"
VALUE=\"mul\"> multiply<br>
<INPUT TYPE=\"radio\" NAME=\"calc\"
VALUE=\"div\"> divide</br>
<INPUT TYPE=\"submit\" NAME=\"submit\"
VALUE=\"Calculate\">
</FORM>
";
if ($submit) {
if ($calc == "add") {$r = $val1 + $val2;}
else if ($calc == "sub") {$r= $val1-$val2;}
else if ($calc == "mul") {$r = $val1*$val2;}
else if ($calc == "div") {$r = $val1/$val2;}
echo "The result of the calculation is: $r";
} else {
echo "$formstring";
}
?>
```

Figure 14 Utilizing the sdame code both for HTML-form generation and data processing in CGI-script mode.

J. Cookies

A cookie is a piece of data stored in the client computer. When a request is sent to a server to get an HTML file, some cookies may be transmitted with that request. The server may send different data depending on the information retrieved from the user. Furthermore, JavaScript is also capable of browsing through all the cookies stored by the user machine [**19**]. This information may be used to enhance performance, for example by remembering the user's preferences. This very useful feature, however, is sometimes abused by some Internet providers, who can spy on the user by analyzing what kinds of web pages are being used.

IV. CONCLUSION

Given limited time and space that was allocated to this tutorial most of the important programming tools that can be applied to solving engineering problems were discussed. Client-server architecture and the system partitioning that were discussed in the introductory sections must be applied to a particular problem. Then based on need one or more tools has to be selected to implement client and server. HTML and JavaScript is generated on the server but utilized on the client side. CGI and ASP with PERL and PHP are stored and utilized on the server. Java can be used on the client side as well as on the server side. It allows implementing a complex functionality of a larger program by using object oriented and well-structured language.

If you are interested in more detailed examples or would like to participate in a 45 hour course offered by Bradley University as a long distance course please visit the Web site that is located at:
http://cegt201.bradley.edu/~olekmali/courses/
and follow the EE-WEB-2000 link to the course materials.

V. REFERENCES

For more information on particular topics discussed in this tutorial please refer to the following source materials that are recommended by the authors:

[1] Kaplan, G., “Ethernet's winning ways,” *IEEE Spectrum,* January 2001, pp. 113-115.

[2] Jamsa K., Lalani S., Weakley S., *Web Programming,* Jamsa Press, Las Vegas, NV, 1996.

[3] Goodman, D., *Dynamic HTML, The Definitive Reference,* O'Reilly & Associates, Sebastopol, CA, 1997.

[4] Flanagan D., *JavaScript, The Definitive Guide,* O'Reilly & Associates, Sebastopol, CA, 1997.

[5] Van der Linden P., *Not Just Java,* Prentice Hall and Sun Microsystems, Palo Alto, CA, 1998.

[6] Web Page: Hank Shiffman, Boosting Java Performance: Native Code and JIT Compilers, http://www.disordered.org/Java-JIT.html, posted in 1998, last time visited in 2001.

[7] Van der Linden P., *Just Java 2,* Prentice Hall and Sun Microsystems, Palo Alto, CA, 1998.

[8] Web Page: Hank Shiffman, Making Sense of Java, http://www.disordered.org/Java-QA.html, posted in 1998, last time visited in 2001.

[9] Hall, M., Brown, L., *Core Web Programming 2nd ed.,* Prentice Hall, Upper Saddle River, NJ, 2001.

[10] Harold, E. R., *Java Network Programming,* O'Reilly, Sebastopol, CA, 1997.

[11] Roff, J.T., ADO: *ActiveX Data Objects,* O'Reilly & Associates, Sebastopol, CA, 2001.

[12] Object management Group, The Common Object Request Broker: Architecture and Specification, v. 2.2, published by Object Management Group, February 1998.

[13] Object management Group Web Site http://www.corba.org/, posted in 1997, visited in 2001.

[14] Thai, T.L., Oram, A., *Learning Dcom,* O'Reilly & Associates, Sebastopol, CA, 1999.

[15] Guelich, S., *CGI Programming with PERL, 2nd ed.,* O'Reilly & Associates, Sebastopol, CA, 2000.

[16] Wall L., Christiansen, T., Orwant, J., *Programming PERL, 3rd ed.,* O'Reilly & Associates, 1996.

[17] Holzner, S., *PERL Black Book,* Coriolis Group, 1999.

[18] Atkinson, L., Core PHP Programming: Using PHP to Build Dynamic Web Sites, 2nd ed., Prentice Hall, Upper Saddle River, NJ, 2000.

[19] CookieCentral.Com, Cookie Central, URL: http://www.cookiecentral.com/, posted in 1996, visited in 2001.