

**KANNUR UNIVERSITY**

SCHOOL OF DISTANCE EDUCATION



SELF INSTRUCTIONAL MATERIAL ON

SDE2B05BCA

WEB TECNOLOGY AND JAVA PROGRAMMING

## **Module I**

### **1.1 Introduction to Internet and Web**

#### **1.1.1. What is the Internet?**

The Internet is a huge collection of computers around the world. These computers are all linked together, and they can "talk" to each other, sharing information. If your computer is connected to the Internet, it can connect to millions of other computers, in many different parts of the world. The popular term for the Internet is the "information highway". Rather than moving through geographical space, it moves your ideas and information through cyberspace – the space of electronic movement of ideas and information.

#### **1.1.2. What can you do on the Internet?**

- Email
- Obtain information.
- Entertainment such as games, radio, reviews of movies.
- Discussion Groups (chat rooms)
- Online Shopping
- Services such as online banking

#### **1.1.3. What do you need to connect to the Internet?**

- Personal Computer
- Modem
- Phone Line
- Internet Service Provider
- Web Browser

#### **1.1.4. History of Internet**

The Internet was the result of some visionary thinking by people in the early 1960s who saw great potential value in allowing computers to share information on research and development in scientific and military fields. J.C.R. Licklider of MIT first proposed a global network of computers in 1962, and moved over to the Defense Advanced Research Projects Agency (DARPA) in late 1962 to head the work to develop it. Leonard Kleinrock of MIT and later UCLA developed the theory of packet switching, which was to form the basis of Internet connections. Lawrence Roberts of MIT connected a Massachusetts computer with a California computer in 1965 over dial-up telephone lines. It showed the feasibility of wide area networking, but also showed that the telephone line's circuit switching was inadequate. Kleinrock's packet switching theory was confirmed. Roberts moved over to DARPA in 1966 and developed his plan for ARPANET. These visionaries and many more left unnamed here are the real founders of the Internet. The Internet, then known as ARPANET, was brought online in 1969 under a contract let by the renamed

Advanced Research Projects Agency (ARPA) which initially connected four major computers at universities in the southwestern US (UCLA, Stanford Research Institute, UCSB, and the University of Utah). The contract was carried out by BBN of Cambridge, MA under Bob Kahn and went online in December 1969. By June 1970, MIT, Harvard, BBN, and Systems Development Corp (SDC) in Santa Monica, Cal. were added. By January 1971, Stanford, MIT's Lincoln Labs, Carnegie-Mellon, and Case-Western Reserve U were added. In months to come, NASA/Ames, Mitre, Burroughs, RAND, and the U of Illinois plugged in. After that, there were far too many to keep listing here.

#### **1.1.5. World Wide Web**

The World Wide Web (abbreviated as WWW or W3, commonly known as the Web), is a system of interlinked hypertext documents accessed via the Internet. With a web browser, one can view web pages that may contain text, images, videos, and other multimedia, and navigate between them via hyperlinks. The Web (World Wide Web) consists of information organized into Web pages containing text and graphic images. A collection of linked Web pages that has a common theme or focus is called a Web site. A collection of linked Web pages that has a common theme or focus is called a Web site.

#### **1.1.6. Internet Service Provider (ISP)**

A commercial organization with permanent connection to the Internet that sells temporary connections to subscribers.

#### **1.1.7. How to access the Web?**

Once you have your Internet connection, then you need special software called a browser to access the Web. Web browsers are used to connect you to remote computers, open and transfer files, display text and images. Web browsers are specialized programs. Examples of Web browser: Netscape Navigator (Navigator) and Internet Explorer.

#### **1.1.8 Client/Server Structure of the Web**

Web is a collection of files that reside on computers, called Web servers that are located all over the world and are connected to each other through the Internet. When you use your Internet connection to become part of the Web, your computer becomes a Web client in a worldwide client/server network. A Web browser is the software that you run on your computer to make it work as a web client.

#### **1.1.9 Hypertext Markup Language (HTML)**

The public files on the web servers are ordinary text files, much like the files used by word-processing software. To allow Web browser software to read them, the text must be formatted according to a generally accepted standard. The standard used on the web is Hypertext markup language (HTML).

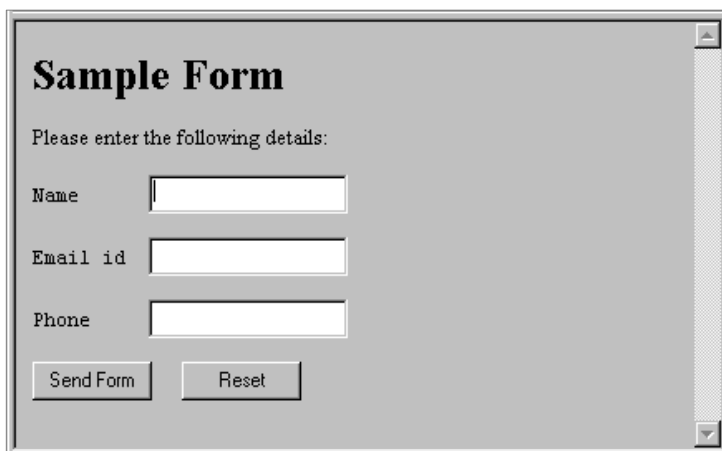
## 1.2 An overview of internet programming

In these classes, we present a number of powerful software technologies that will enable you to build systems that can integrate Internet and web components, and remote databases. We present the “client-side” and “server-side” of web programming. For the client side we present a carefully paced introduction to using the popular JavaScript language and the closely related technologies of XHTML (Extensible Hyper Text Markup Language), CSS (Cascading Style Sheets) and the DOM (Document Object Model). The third class concentrates on using technologies such as web servers, databases (integrated collections of data), PHP, Ruby on Rails, ASP.NET, ASP.NET Ajax and Java Server Faces (JSF) to build the server side of web-based applications. These portions of applications typically run on “heavy-duty” computer systems on which organizations’ business-critical websites reside. By mastering the technologies in these courses, you’ll be able to build substantial web-based, client/server, database-intensive, “multitier” applications.

Internet applications are client-server applications, and can be split into two pieces:

1. Forms
2. Server-side programs

The form is the part your end-user sees. It is displayed in a Web browser, and provides controls through which your end-user can enter data. The picture below shows you a simple form:



The image shows a web browser window with a title bar. Inside the window, the page has a heading "Sample Form" in bold. Below the heading is a text prompt: "Please enter the following details:". There are three text input fields arranged vertically. The first field is labeled "Name", the second is labeled "Email id", and the third is labeled "Phone". Below these fields are two buttons: "Send Form" and "Reset". The browser window has a vertical scrollbar on the right side.

When the end-user clicks the Send Form button, the information on the form is packaged, and sent to a server-side program. The server-side program only runs when it is started from a form, or from a link on a Web page. The server-side program processes the information on the form, and returns a result to the end-user.

Depending on what the program does, the result is displayed on another form, or perhaps a simple HTML page showing text.

## 1.3 Introduction to HTML

HyperText Markup Language (HTML) is a markup language used to create the content and semantic structure of Web pages. A Web page is comprised of a number of HTML elements, each

of which has a particular meaning in the context of a Web page. Some elements are stand-alone, while others can be nested to create increasingly complex structure for your content. Web browsers interpret HTML to build the content of a page, and interpret that HTML in the context of Cascading Style Sheets (CSS) that affect the visual appearance of that content.

### **1.3.1 A brief history of HTML**

HTML was originally developed by Tim Berners-Lee while at CERN, and popularized by the Mosaic browser developed at NCSA. During the course of the 1990s it has blossomed with the explosive growth of the Web. During this time, HTML has been extended in a number of ways. The Web depends on Web page authors and vendors sharing the same conventions for HTML. This has motivated joint work on specifications for HTML.

Most people agree that HTML documents should work well across different browsers and platforms. Achieving interoperability lowers costs to content providers since they must develop only one version of a document. If the effort is not made, there is much greater risk that the Web will devolve into a proprietary world of incompatible formats, ultimately reducing the Web's commercial potential for all participants.

Each version of HTML has attempted to reflect greater consensus among industry players so that the investment made by content providers will not be wasted and that their documents will not become unreadable in a short period of time.

HTML has been developed with the vision that all manner of devices should be able to use information on the Web: PCs with graphics displays of varying resolution and color depths, cellular telephones, hand held devices, devices for speech for output and input, computers with high or low bandwidth, and so on.

### **1.3.2 What is HTML?**

HTML is a language for describing web pages.

1. HTML stands for Hyper Text Markup Language
2. HTML is not a programming language, it is a markup language
3. A markup language is a set of markup tags
4. The purpose of the tags are to describe page content

### **1.4 Introduction to the structure of an HTML document**

An HTML 4 document is composed of three parts:

- A line containing HTML version information,
- A declarative header section (delimited by the HEAD element), contains title and meta data of a web document.
- A body, which contains the document's actual content which are displayed by web browser. The body may be implemented by the BODY element or the FRAMESET element.

White space (spaces, newlines, tabs, and comments) may appear before or after each section. Here's an example of a simple HTML document:

Example

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
"http://www.w3.org/TR/html4/strict.dtd">
<HTML>
  <HEAD>
    <TITLE>My first HTML document</TITLE>
  </HEAD>
  <BODY>
    <P>Hello world!
  </BODY>
</HTML>
```

## 1.5 HTML Elements

"HTML tags" and "HTML elements" are often used to describe the same thing.

But strictly speaking, an HTML element is everything between the start tag and the end tag, including the tags:

- An HTML element starts with a start tag / opening tag
- An HTML element ends with an end tag / closing tag
- The element content is everything between the start and the end tag
- Some HTML elements have empty content
- Empty elements are closed in the start tag
- Most HTML elements can have attributes

*Elements* are the structures that describe parts of an HTML document. For example, the P element represents a *paragraph* while the EM element gives *emphasized* content.

The basic unit of information in HTML is conveyed by the element. Elements basically answer the question "what kind of information is this?" and define the semantic meaning of their content. Some elements have very precise meaning, as in "this is an image," "this is a header," or "this is an ordered list," while some are less specific as in "this is a part of the page" or "this is a part of the text", and yet others are used for technical reasons. But in some way or the other they all have a semantic value.

Most elements may contain other elements, forming a hierarchic structure, a tree, called the DOM - the Document Object Model.

An element has three parts: a start tag, content, and an end tag.

### 1.5.1 Empty HTML Elements

HTML elements with no content are called empty elements.

`<br>` is an empty element without a closing tag (the `<br>` tag defines a line break).

## 1.6 HTML tags

HTML markup tags are usually called HTML tags

- HTML tags are keywords (tag names) surrounded by angle brackets like `<html>`
- HTML tags normally come in pairs like `<b>` and `</b>`
- The first tag in a pair is the start tag, the second tag is the end tag
- The end tag is written like the start tag, with a forward slash before the tag name
- Start and end tags are also called opening tags and closing tags

General syntax

`<tagname>content</tagname>`

A *tag* is special text--"markup"--that is delimited by "<" and ">". An end tag includes a "/" after the "<". For example, the EM element has a start tag, `<EM>`, and an end tag, `</EM>`. The start and end tags surround the *content* of the EM element:

`<EM>This is emphasized text</EM>`

### 1.6.1 Starting and ending tags

HTML uses plain text as a foundation and attaches special meaning to anything that starts with the less than sign (<) and ends with the greater than sign (>). Such markup is called a tag. Here is a simple example:

`<p>This is text within a paragraph.</p>`

In that example there is a *start* tag and a *closing* tag. Closing tags use the same tag name as the starting tag, but also contain a *forward slash* immediately after the leading less than sign. *Most* elements in HTML are written using both start and closing tags. Start and closing tags should be properly nested, that is closing tags should be written in the opposite order of the start tags. Proper nesting is one rule that must be obeyed in order to write valid code.

This is an example of *valid* code:

`<em>I <strong>really</strong> mean that</em>.`

Element names are always case-insensitive, so `<em>`, `<eM>`, and `<EM>` are all the same.

Elements cannot overlap each other. If the start tag for an EM element appears within a P, the EM's end tag must also appear within the same P element.

Some elements allow the start or end tag to be omitted. For example, the LI end tag is always optional since the element's end is implied by the next LI element or by the end of the list:

`<UL>`

`<LI>First list item; no end tag`

`<LI>Second list item; optional end tag included</LI>`

```
<LI>Third list item; no end tag
</UL>
```

Some elements have no end tag because they have no content. These elements, such as the BR element for line breaks, are represented only by a start tag and are said to be *empty elements*.

## 1.7 Attributes

The start tag may contain additional information. Such information is called an attribute. Attributes give additional meaning to an element. An element's *attributes* define various properties for the element. For example, the IMG element takes a SRC attribute to provide the location of the image and an ALT attribute to give alternate text for those not loading images:

```
<IMG SRC="wdglogo.gif" ALT="Web Design Group">
```

Attributes usually consist of 2 parts:

- An attribute name.
- An attribute value.

An attribute is included in the start tag only--never the end tag--and takes the form

```
Attribute-name="Attribute-value".
```

The attribute value is delimited by single or double quotes. The quotes are optional if the attribute value consists solely of letters in the range A-Z and a-z, digits (0-9), hyphens ("-"), periods ("."), underscores ("\_"), and colons (":"). Attribute names are case-insensitive, but attribute values may be case-sensitive.

## 1.8 HTML Documents = Web Pages

- HTML documents describe web pages
- HTML documents contain HTML tags and plain text
- HTML documents are also called web pages

## 1.9 Comments

Comments in HTML have a complicated syntax that can be simplified by following this rule: Begin a comment with "<!--", end it with "-->", and do not use "--" within the comment.

```
<!-- An example comment -->

HTML Document Example

<!DOCTYPE html>

<html>

<body>

<p>This is my first paragraph.</p>

</body>

</html>
```

The example above contains 3 HTML elements.



## HTML Example Explained

*The <p> element:*

```
<p>This is my first paragraph.</p>
```

*The <p> element defines a paragraph in the HTML document.*

*The element has a start tag <p> and an end tag </p>.*

*The element content is: This is my first paragraph.*

*The <body> element:*

```
<body>
```

```
<p>This is my first paragraph.</p>
```

```
</body>
```

*The <body> element defines the body of the HTML document.*

*The element has a start tag <body> and an end tag </body>.*

*The element content is another HTML element (a p element).*

*The <html> element:*

```
<html>
```

```
<body>
```

```
<p>This is my first paragraph.</p>
```

```
</body>
```

```
</html>
```

The <html> element defines the whole HTML document. The element has a start tag <html> and an end tag </html>. The element content is another HTML element (the body element).

### 1.10 HTML <html> Tag

Example

A simple HTML document, with the minimum of required tags:

```
<html>
```

```
<head> <title>HTML 4.01 Tag Reference</title> </head>
```

```
<body> The content of the document.....</body>
```

```
</html>
```

#### 1.10.1 Definition and Usage

The <html> tag tells the browser that this is an HTML document. The <html> element is also known as the root element. The <html> tag is the container for all other HTML elements (except for the<!DOCTYPE> tag).

### 1.11 HTML <title> Tag

Define a title for your HTML document:

Example

```
<html>
<head> <title>HTML 4.01 Tag Reference</title> </head>
<body> The content of the document.....</body>
</html>
```

### 1.11.1 Definition and Usage

The <title> tag is required in all HTML documents and it defines the title of the document.

The <title> element:

- defines a title in the browser toolbar
- provides a title for the page when it is added to favorites
- displays a title for the page in search-engine results

## 1.12 HTML <body> Tag

### 1.12.1 Definition and Usage

The <body> tag defines the document's body. The <body> element contains all the contents of an HTML document, such as text, hyperlinks, images, tables, lists, etc.

### 1.12.2 Attributes

*background* - Specifies a background image for a document

*bgcolor* - Specifies the background color of a document

The <body> tag supports the following standard attributes:

| Attribute | Value                   | Description  |
|-----------|-------------------------|--|
| class     | <i>classname</i>        | Specifies a classname for an element                       |
| dir       | rtl                     | Specifies the text direction for the content in an element |
|           | ltr                     |  |
| id        | <i>id</i>               | Specifies a unique id for an element                       |
| lang      | <i>language_code</i>    | Specifies a language code for the content in an element    |
| style     | <i>style_definition</i> | Specifies an inline style for an element                   |
| title     | <i>text</i>             | Specifies extra information about an element               |

### 1.12.3 Event Attributes

The <body> tag supports the following event attributes:

| Attribute  | Value         | Description                              |
|------------|---------------|--|
| onclick    | <i>script</i> | Script to be run on a mouse click        |
| ondblclick | <i>script</i> | Script to be run on a mouse double-click |
| onload     | <i>script</i> | Script to be run when a document load    |

|             |               |   |
|-------------|---------------|---|
| onmousedown | <i>script</i> | Script to be run when mouse button is pressed               |
| onmousemove | <i>script</i> | Script to be run when mouse pointer moves                   |
| onmouseout  | <i>script</i> | Script to be run when mouse pointer moves out of an element |
| onmouseover | <i>script</i> | Script to be run when mouse pointer moves over an element   |
| onmouseup   | <i>script</i> | Script to be run when mouse button is released              |
| onkeydown   | <i>script</i> | Script to be run when a key is pressed                      |
| onkeypress  | <i>script</i> | Script to be run when a key is pressed and released         |
| onkeyup     | <i>script</i> | Script to be run when a key is released                     |
| onunload    | <i>script</i> | Script to be run when a document unload                     |

### 1.13 Physical Style Tags

There are nine physical styles provided by the current HTML and XHTML standards, including bold, italic, monospaced, underlined, strike-through, larger, smaller, superscripted, and subscripted text. As we discuss each physical tag in detail, keep in mind that they convey an acute styling for the immediate text.

#### 1.13.1 The <b> Tag

The <b> tag is the physical equivalent of the <strong> content-based style tag, but without the latter's extended meaning. The <b> tag explicitly boldfaces a character or segment of text that is enclosed between it and its corresponding (</b>) end tag. If a boldface font is not available, the browser may use some other representation, such as reverse video or underlining.

#### 1.13.2 The <big> Tag

The <big> tag makes it easy to increase the size of text. It couldn't be simpler: the browser renders the text between the <big> tag and its matching </big> ending tag one font size larger than the surrounding text. If that text is already at the largest size, <big> has no effect.

Even better, you can nest <big> tags to enlarge the text. Each <big> tag makes the text one size larger, up to a limit of size seven, as defined by the font model.

Be careful with your use of the <big> tag, though. Because browsers are quite forgiving and try hard to understand a tag, those that don't support <big> often interpret it to mean bold.

#### 1.13.3 The <blink> Tag

Text contained between the <blink> tag and its end tag </blink> does just that: blink on and off. Netscape for Macintosh, for example, simply and reiteratively reverses the background and foreground colors for the <blink> enclosed text.

We cannot effectively reproduce the animated effect in these static pages, but it is easy to imagine and best left to the imagination, too. That's because blinking text has two primary effects: it gets your reader's attention, and then promptly annoys them to no end. Blinking text should be used sparingly in any context.

#### **1.13.4 The `<i>` Tag**

The `<i>` tag is like the `<em>` content-based style tag. It and its necessary (`</i>`) end tag tell the browser to render the enclosed text in an italic or oblique typeface. If the typeface is not available to the browser, highlighting, reverse video, or underlining might be used.

#### **1.13.5 The `<s>` Tag**

The `<s>` tag is an abbreviated form of the `<strike>` tag supported by both Internet Explorer and Netscape. It is now a deprecated tag in HTML 4 and XHTML, meaning don't use it; it will eventually go away.

#### **1.13.6 The `<small>` Tag**

The `<small>` tag works just like its `<big>` counterpart (see previous description), except it decreases the size of text instead of increasing it. If the enclosed text is already at the smallest size supported by the font model, `<small>` has no effect.

Like `<big>`, you may also nest `<small>` tags to sequentially shrink text. Each `<small>` tag makes the text one size smaller than the containing `<small>` tag, down to a limit of size one.

#### **1.13.7. The `<strike>` Tag**

Most browsers will put a line through ("strike through") text that appears inside the `<strike>` tag and its `</strike>` end tag. Presumably, it is an editing markup that tells the reader to ignore the text passage, reminiscent of the days before typewriter correction tape. You'll rarely, if ever, see the tag in use today, and probably never will: it is deprecated in HTML 4 and XHTML, just one version away from complete elimination from the standard.

#### **1.13.8. The `<sub>` Tag**

The text contained between the `<sub>` tag and its `</sub>` end tag gets displayed half a character's height lower, but in the same font and size as the current text flow. Both `<sub>` and its `<sup>` counterpart is useful for math equations and in scientific notation, as well as with chemical formula.

#### **1.13.9. The `<sup>` Tag**

The `<sup>` tag and its `</sup>` end tag superscripts the enclosed text; it gets displayed half a character's height higher, but in the same font and size as the current text flow. This tag is useful for adding footnotes to your documents, along with exponential values in equations.

### 1.13.10. The <tt> Tag

In a manner like the <code> and <kbd> tags, the <tt> tag and necessary </tt> end tag direct the browser to display the enclosed text in a monospaced typeface. For those browsers that already use a monospaced typeface, this tag may make no discernible change in the presentation of the text.

### 1.13.11. The <u> Tag

This tag tells the browser to underline the text contained between the <u> and the corresponding </u> tag. The underlining technique is simplistic, drawing the line under spaces and punctuation as well as the text. This tag is deprecated in HTML 4 and XHTML and will be eliminated in the next version of the standard.

### 1.13.12. The dir and lang Attributes

The dir attribute lets you advise the browser as to which direction the text within the physical tag ought to be displayed, and lang lets you specify the language used within the tag.

### 1.13.13. The class, id, style, and title Attributes

Although each physical tag has a defined style, you can override that style by defining your own look for each tag. This new look can be applied to the physical tags using either the style or class attributes.

You also may assign a unique id to the physical style tag, as well as a less rigorous title, using the respective attribute and accompanying quote-enclosed string value.

### 1.13.14. Event Attributes

Like with content-based style tags, user-initiated mouse and keyboard events can happen in and around a physical-style tag's contents. Many of these events are recognized by the browser if it conforms to current standards, and, with the respective "on" attribute and value, you may react to the event by displaying a user dialog box or activating some multimedia event.

### 1.13.15. Summary of Physical Style Tags

The various graphical browsers render text inside the physical style tags in similar fashion. Table 4-2 summarizes these browser's display styles for the native tags. Style sheet definitions may override these native display styles.

| Tag           | Meaning                                 | Display Style     |
|---------------|---|-------------------|
| <b>           | Bold contents                           | bold              |
| <big>         | Increased font size                     | bigger text       |
| <blink>       | Alternating fore- and background colors | blinking text     |
| <i>           | Italic contents                         | <i>italic</i>     |
| <small>       | Decreased font size                     | smaller text      |
| <s>, <strike> | Strike-through text                     | <del>strike</del> |

|                          |                      |             |
|--------------------------|----------------------|-------------|
| <code>&lt;sub&gt;</code> | Subscripted text     | subscript   |
| <code>&lt;sup&gt;</code> | Superscripted text   | superscript |
| <code>&lt;tt&gt;</code>  | Teletypewriter style | monospaced  |
| <code>&lt;u&gt;</code>   | Underlined contents  | underlined  |

#### 1.13.16. HTML Lines

The `<hr />` tag creates a horizontal line in an HTML page. The `hr` element can be used to separate content:

#### 1.13.17. HTML Line Breaks

Use the `<br />` tag if you want a line break (a new line) without starting a new paragraph:

Example

```
<p>This is<br />a para<br />graph with line breaks</p>
```

#### 1.13.18. HTML Headings

Headings are defined with the `<h1>` to `<h6>` tags.

`<h1>` defines the most important heading. `<h6>` defines the least important heading.

Example

```
<h1>This is a heading</h1>
```

```
<h2>This is a heading</h2>
```

```
<h3>This is a heading</h3>
```

#### 1.13.19. HTML Paragraphs

HTML Paragraphs Paragraphs are defined with the `<p>` tag.

Example

```
<p>This is a paragraph</p>
```

```
<p>This is another paragraph</p>
```

### 1.14. HTML Lists

HTML supports five types of lists. A list is first marked with the start and end list tag and then each list item is indicated with a list item tag `<LI>` (unless it is a definition list). List item tags have end tags, but they are optional since a new list item tag implies the end of the previous item. Lists may be nested and if they are nested lists are indented farther than their parent list when displayed. Any type of list can have a `compact` attribute, which forces the list items closer together both horizontally and vertically.

#### 1.14.1. Unordered List: `<UL>`

A list of items which may appear in any particular order. It is usually displayed as a bulleted list of items. An unordered list starts with the `<ul>` tag. Each list item starts with the `<li>` tag.

The list items are marked with bullets (typically small black circles).

```

<ul>
<li>Coffee</li>
<li>Milk</li>
</ul>

```

How the HTML code above looks in a browser:

- Coffee
- Milk

#### Attributes

| Attribute | Value  | Description                                     |
|-----------|--------|---|
|           | disc   |   |
| type      | square | Specifies the kind of marker to use in the list |
|           | circle |   |

#### 1.14.2. Ordered List: <OL>

A list of items to be displayed in a particular order. These are usually numbered when displayed. An ordered list starts with the <ol> tag. Each list item starts with the <li> tag.

The list items are marked with numbers.

```

<ol>
<li>Coffee</li>
<li>Milk</li>
</ol>

```

How the HTML code above looks in a browser:

1. Coffee
2. Milk

#### Attributes

| Attribute | Value         | Description                                     |
|-----------|---------------|---|
| start     | <i>number</i> | Specifies the start value of an ordered list    |
| type      | 1, A, a, I, i | Specifies the kind of marker to use in the list |

#### 1.14.3. Definition List

A definition list <DL> is a list of terms <DT> and their definitions <DD>. Each definition is usually displayed indented slightly in relation to the term. Each term should have a corresponding definition. A definition list is a list of items, with a description of each item. The <dl> tag defines a definition list. The <dl> tag is used in conjunction with <dt> (defines the item in the list) and <dd> (describes the item in the list):

```

<dl>
<dt>Coffee</dt>
<dd>- black hot drink</dd>
<dt>Milk</dt>
<dd>- white cold drink</dd>
</dl>

```

How the HTML code above looks in a browser:

*Coffee*

- black hot drink

*Milk*

- white cold drink

### 1.15. The HTML Image Tag

The image tag is used to place an image on the web page. In its most simple form it looks like this:

```

```

It is very important to understand that images are not technically "part" of the web page file, they are separate files which are inserted into the page when it is viewed by a browser. So a simple web page with one image is actually two files - the HTML file and the image file. The above example illustrates this. In this example the two files are both located in the same folder. The HTML file includes an image tag which refers to *image1.jpg*. When the HTML file is displayed in a browser, it requests the image file and places it on the page where the tag appears.

#### 1.15.1. Attributes

As you can see, the most important attribute of the image tag is *src*, which means *source* and tells the browser where the image file is. The major attributes are listed below

| Attribute | Value               | Description   |
|-----------|---------------------|---|
| alt       | <i>text</i>         | Specifies an alternate text for an image                              |
| src       | <i>URL</i>          | Specifies the URL of an image   |
|           | Top, bottom         | Deprecated. Use styles instead.                                       |
| align     | middle, left, right | Specifies the alignment of an image according to surrounding elements |
|           |                     | Deprecated. Use styles instead.                                       |
| border    | <i>pixels</i>       | Specifies the width of the border around an image                     |
|           | <i>pixels</i>       |   |
| height    | %                   | Specifies the height of an image                                      |
|           |                     |   |
| hspace    | <i>pixels</i>       | Deprecated. Use styles instead.                                       |



|          |                    |  |
|----------|--------------------|--|
|          |                    | Specifies the whitespace on left and right side of an image  |
| ismap    | ismap              | Specifies an image as a server-side image-map  |
| longdesc | <i>URL</i>         | Specifies the URL to a document that contains a long description of an image                           |
| usemap   | <i>#mapname</i>    | Specifies an image as a client-side image-map  |
| vspace   | <i>pixels</i>      | Deprecated. Use <code>styles</code> instead.<br>Specifies the whitespace on top and bottom of an image |
| width    | <i>pixels</i><br>% | Specifies the width of an image  |

#### Example

```
<!DOCTYPE html>
<html>
<body>

</body>
</html>
```

### 1.16. HTML <a> Tag

The HTML *anchor* element is used to create a link to a resource (another web page, a file, etc.) or to a specific place within a web page. The anchor tag is written like this:

```
<a>
```

The anchor tag alone won't do anything without an attribute and value, so let's look at the attributes we can use.

The <a> tag defines an anchor. An anchor can be used in two ways:

1. To create a link to another document, by using the href attribute
2. To create a bookmark inside a document, by using the name attribute

The <a> element is usually referred to as a link or a hyperlink. The most important attribute of the <a> element is the href attribute, which indicates the link's destination.

By default, links will appear as follows in all browsers:

- An unvisited link is underlined and blue
- A visited link is underlined and purple
- An active link is underlined and red

#### 1.16.1. Attributes

| Attribute | Value                | Description                                      |
|-----------|----------------------|--|
| charset   | <i>char_encoding</i> | Specifies the character-set of a linked document |

|          |   |   |
|----------|---|---|
| coords   | <i>coordinates</i>  | Specifies the coordinates of a link   |
| href     | <i>URL</i>  | Specifies the URL of the page the link goes to                                  |
| hreflang | <i>language_code</i>  | Specifies the language of the linked document                                   |
| name     | <i>section_name</i>   | Specifies the name of an anchor   |
| rel      | <i>text</i>   | Specifies the relationship between the current document and the linked document |
| rev      | <i>text</i>   | Specifies the relationship between the linked document and the current document |
| shape    | Default, rect<br>circle, poly<br>_blank<br>_parent<br>_self<br>_top<br><i>framename</i> | Specifies the shape of a link   |
| tabindex | <i>number</i>   | Specifies the tab order of an element   |
| title    | <i>text</i>   | Specifies extra information about an element                                    |

#### Example

```

<!DOCTYPE html>
<html>
<body>
<a href="http://www.w3schools.com">Visit W3Schools.com!</a>
</body>
</html>

```

#### 1.16.2. The HREF Attribute

To create a link, you have to know the web address of the file you want to link to, whether it's another web page of your own site, another website, or a link to file such as a PDF document, sound file, or another type of file.

Suppose you want to link to the front page of my site. The web address is

: <http://www.boogiejack.com>. You'd code the link like this:

```
<a href="http://www.boogiejack.com">Boogie Jack</a>
```

The *href* part, shown in dark blue text, is short for *hypertext reference*. This is the attribute that defines the address of the file you want to link to.

The equal sign always connects an attribute to the attribute's value. So in this case, *href* is the attribute, and *http://www.boogiejack.com* is the value. The value is always enclosed in quotation marks.

The *Boogie Jack* part, shown in green text, is the anchor text, or sometimes called the link text. This is the part of a link that is clickable.

If you link to a page on another site you need to use the full web address as shown in the example above. If you're linking to a different page on your own site you only need to use the page name and extension if the page is kept in the same directory.

For example, suppose you want to link to a page you've saved with the name of *MyPage.html*. You'd code it like this:

```
<a href="MyPage.html">My Page</a>
```

By linking to your own internal pages *without* using the full web address your pages will load faster. If you use the full web address the browser goes back out to the Internet to find your site all over again, which takes longer. If you don't use the full path the browser only checks on your site for the file.

File names, which includes the name of the web page and the extension, are case sensitive. That means you must use the same capitalization in the web address of the file that was used when the file was saved.

### 1.16.2. The NAME Attribute

The *name* attribute allows an anchor tag to be used to point to a specific place on a web page. You might link from the bottom of a long page to the top of the page, or link from an item in a Table of Contents to the corresponding item where it appears on the page.

The syntax for using the name attribute is like this:

```
<a name="top"></a> » or...
```

```
<a name="TOC">Table of Contents</a>
```

You can leave out the text between the "a" tags or use them to surround some text. The appearance of the text won't change unless you have defined a hover color for your links. If you have, then the text will change to the hover color when a user's cursor is on it. It will not be clickable, however, because this is not the link, this is the anchor a link will point to.

In the first example you would link to the top of a page from the bottom of a long page, and maybe other points in between so your visitors could jump back to the top instead of scrolling.

Or, you might place a named anchor as shown in the second example around the Table of Contents for an online newsletter, for example, then link to it from strategic places down the page so your visitors can quickly jump to the Table of Contents after reading an article.

For the newsletter example, to link to that named anchor you'd code your link like this:

`<a href="#TOC">Table of Contents</a>`

As you can see, it's simply a hash mark (#) in front of the actual anchor name. The hash mark tells the browser the link is on the current page.

You can also link to a named anchor on another page. The syntax for that is:

`<a href="AnotherPage.html#name">Link Text</a>`

That would be a page located in the same directory as the current page. You can link to anchor points on other sites (if they have an anchor point) by including the full web address of the page.

As you probably noticed, it's a normal link followed by the hash mark and the anchor name. That tells the browser to go to the other page, then to find the named anchor on that page.

### 1.16.3. The TARGET Attribute

The *target* attribute allows you to determine where the link will open. With a framed site, it allows you to target a link to a specific frame. The most common use is to have off-site links open in a new browser window.

Here's how to open a link in a new window:

`<a href="http://www.site.com" target="_blank">Link Text</a>`

By adding the part in green to a link, the link will open in a new window or a new tab, depending on the browser in use and how it's configured.

Syntax

`<a target="_blank|_self|_parent|_top|framename">`

| Value                  | Description   |
|------------------------|---|
| <code>_blank</code>    | Opens the linked document in a new window or tab                                |
| <code>_self</code>     | Opens the linked document in the same frame as it was clicked (this is default) |
| <code>_parent</code>   | Opens the linked document in the parent frame                                   |
| <code>_top</code>      | Opens the linked document in the full body of the window                        |
| <code>framename</code> | Opens the linked document in a named frame                                      |

## 1.7 HTML Table

Tables are defined with the `<table>` tag. A table is divided into rows (with the `<tr>` tag), and each row is divided into data cells (with the `<td>` tag). `td` stands for "table data," and holds the content of a data cell. A `<td>` tag can contain text, links, images, lists, forms, other tables, etc. A more complex HTML table may also include `<caption>`, `<col>`, `<colgroup>`, `<thead>`, `<tfoot>`, and `<tbody>` elements.

### 1.7.1. HTML `<table>` Tag

The `<table>` tag defines an HTML table.

```

<!DOCTYPE html>
<html>
<body>
<table border="1">
  <tr>
    <th>Month</th>
    <th>Savings</th>
  </tr>
  <tr>
    <td>January</td>
    <td>$100</td>
  </tr>
  <tr>
    <td>February</td>
    <td>$80</td>
  </tr>
</table>
</body>
</html>

```

The <table> tag is supported in all major browsers.

### Attributes

| Attribute   | Value                         | Description  |
|-------------|-------------------------------|--|
| bgcolor     | <i>rgb(x,x,x)</i>             | Deprecated. Use styles instead.                                    |
|             | <i>#xxxxxx</i>                |  |
|             | <i>colorname</i>              | Specifies the background color for a table                         |
| border      | <i>pixels</i>                 | Specifies the width of the borders around a table                  |
| cellpadding | <i>pixels</i>                 | Specifies the space between the cell wall and the cell content     |
| cellspacing | <i>pixels</i>                 | Specifies the space between cells                                  |
| rules       | None, groups, rows, cols, all | Specifies which parts of the inside borders that should be visible |
| width       | <i>pixels</i>                 | Specifies the width of a table                                     |
|             | <i>%</i>                      |  |

### 1.17.2. Table rows - <tr> tag

The <tr> tag defines a row in an HTML table. A <tr> element contains one or more <th> or <td> elements.

```
<tr>
  <th>Month</th>
  <th>Savings</th>
</tr>
```

### 1.17.3. Table Cells - <th> and <td> tags

The <th> tag defines a header cell in an HTML table. The <td> tag defines a standard cell in an HTML table.

An HTML table has two kinds of cells:

- Header cells - contains header information (created with the <th> element)
- Standard cells - contains data (created with the <td> element)

The text in <th> elements are bold and centered by default. The text in <td> elements are regular and left-aligned by default.

```
<th>Month</th>
<td>February</td>
```

#### Attributes

| Attribute | Value   | Description   |
|-----------|---|---|
| abbr      | <i>text</i>                                       | Not supported in HTML5. Specifies an abbreviated version of the content in a cell         |
| align     | Left,<br>center,<br>char<br><br><i>rgb(x,x,x)</i> | right<br>Not supported in HTML5. Aligns the content in a cell                             |
| bgcolor   | <i>#xxxxxx</i><br><i>colorname</i>                | Not supported in HTML5. Deprecated in HTML 4.01. Specifies the background color of a cell |
| char      | <i>character</i>                                  | Not supported in HTML5. Aligns the content in a cell to a character                       |
| colspan   | <i>number</i>                                     | Specifies the number of columns a cell should span  |
| height    | <i>pixels</i><br><i>%</i>                         | Not supported in HTML5. Deprecated in HTML 4.01.<br>Sets the height of a cell             |
| rowspan   | <i>number</i>                                     | Sets the number of rows a cell should span  |

|        |               |        |  |
|--------|---------------|--------|--|
| valign | Top,          | middle | Not supported in HTML5. Vertical aligns the content in a cell                  |
|        | bottom        |        |  |
|        | baseline      |        |  |
| width  | <i>pixels</i> |        | Not supported in HTML5. Deprecated in HTML 4.01. Specifies the width of a cell |
|        | %             |        |  |

#### 1.17.4. Height attribute

The <td> height attribute is not supported in HTML5. Use CSS instead. The height attribute of <td> is deprecated in HTML 4.01. The height attribute specifies the height of a cell. Normally, a cell takes up the space it needs to display the content. The height attribute is used to set a predefined height of a cell.

#### 1.17.5. Width attribute

The <table> width attribute is not supported in HTML5. Use CSS instead. The width attribute specifies the width of a table. If the width attribute is not set, a table takes up the space it needs to display the table data.

#### 1.17.6. Rowspan attribute

The rowspan attribute specifies the number of rows a cell should span.

#### 1.17.7. Colspan attribute

The colspan attribute defines the number of columns a cell should span.

#### 1.17.8. Border attribute

The border attribute specifies if a border should be displayed around the table cells or not. The value "1" indicates borders should be displayed, and that the table is NOT being used for layout purposes.

Example

```
<html>
<body>
<table border=1 width=100 height=150>
<tr><th rowspan=2>name</th><th colspan=3>sub</th></tr>
<tr><th>c</th><th>c++</th><th>java</th></tr>
</table>
</body>
</html>
```

### 1.18 HTML forms

Form is used to collect information from a user. They are used to pass data to a server. An HTML form is a section of a document containing normal content, markup, special elements called controls (checkboxes, radio buttons, menus, etc.), and labels on those controls. Users generally "complete" a

form by modifying its controls (entering text, selecting menu items, etc.), before submitting the form to an agent for processing (e.g., to a Web server, to a mail server, etc.)

```
<form>
  input elements
</form>
```

The `<form>` tag is used to create an HTML form for user input. The `<form>` element can contain one or more of the following form elements:

- `<input>`
- `<textarea>`
- `<button>`
- `<select>`
- `<option>`
- `<optgroup>`
- `<fieldset>`
- `<label>`

Here's a simple form that includes labels, radio buttons, and push buttons (reset the form or submit it):

```
<FORM action="http://somesite.com/prog/adduser" method="post">
  <P>
    <LABEL for="firstname">First name: </LABEL>
      <INPUT type="text" id="firstname"><BR>
    <LABEL for="lastname">Last name: </LABEL>
      <INPUT type="text" id="lastname"><BR>
    <LABEL for="email">email: </LABEL>
      <INPUT type="text" id="email"><BR>
    <INPUT type="radio" name="sex" value="Male"> Male<BR>
    <INPUT type="radio" name="sex" value="Female"> Female<BR>
    <INPUT type="submit" value="Send"> <INPUT type="reset">
  </P>
</FORM>
```

#### 1.18.1. Attributes

| Attribute | Value | Description  |
|-----------|-------|--|
| action    | URL   | Specifies where to send the form-data when a form is submitted |



|        |               |  |
|--------|---------------|--|
| method | get           | Specifies the HTTP method to use when sending form-data                            |
|        | post          |  |
| name   | <i>text</i>   | Specifies the name of a form   |
| target | _blank,       | Specifies where to display the response that is received after submitting the form |
|        | _parent, _top |  |

- The layout of the form (given by the contents of the element).
- The program that will handle the completed and submitted form (the action attribute). The receiving program must be able to parse name/value pairs in order to make use of them.
- The method by which user data will be sent to the server (the method attribute).
- A character encoding that must be accepted by the server in order to handle this form (the accept-charset attribute). User agents may advise the user of the value of the accept-charset attribute and/or restrict the user's ability to enter unrecognized characters.

### 1.18.2. Controls

Users interact with forms through named *controls*. A control's "*control name*" is given by its name attribute. The scope of the name attribute for a control within a FORM element is the FORM element.

Each control has both an initial value and a current value, both of which are character strings. Please consult the definition of each control for information about initial values and possible constraints on values imposed by the control. In general, a control's "*initial value*" may be specified with the control element's value attribute.

The control's "*current value*" is first set to the initial value. Thereafter, the control's current value may be modified through user interaction and scripts.

When a form is submitted for processing, some controls have their name paired with their current value and these pairs are submitted with the form. Those controls for which name/value pairs are submitted are called successful controls.

### Control types

HTML defines the following control types:

#### buttons

Authors may create three types of buttons:

- submit buttons: When activated, a submit button submits a form. A form may contain more than one submit button.
- reset buttons: When activated, a reset button resets all controls to their initial values.

- push buttons: Push buttons have no default behavior. Each push button may have client-side scripts associated with the element's event attributes. When an event occurs (e.g., the user presses the button, releases it, etc.), the associated script is triggered.

We can create buttons with the `BUTTON` element or the `INPUT` element. Please consult the definitions of these elements for details about specifying different button types.

### **checkboxes**

Checkboxes (and radio buttons) are on/off switches that may be toggled by the user. A switch is "on" when the control element's `checked` attribute is set. When a form is submitted, only "on" checkbox controls can become successful.

Several checkboxes in a form may share the same control name. Thus, for example, checkboxes allow users to select several values for the same property. The `INPUT` element is used to create a checkbox control.

### **radio buttons**

Radio buttons are like checkboxes except that when several share the same control name, they are mutually exclusive: when one is switched "on", all others with the same name are switched "off". The `INPUT` element is used to create a radio button control.

If no radio button in a set sharing the same control name is initially "on", user agent behavior for choosing which control is initially "on" is undefined.

### **menus**

Menus offer users options from which to choose. The `SELECT` element creates a menu, in combination with the `OPTGROUP` and `OPTION` elements.

### **text input**

We can create two types of controls that allow users to input text. The `INPUT` element creates a single-line input control and the `TEXTAREA` element creates a multi-line input control. In both cases, the input text becomes the control's current value.

### **file select**

This control type allows the user to select files so that their contents may be submitted with a form. The `INPUT` element is used to create a file select control.

### **hidden controls**

We may create controls that are not rendered but whose values are submitted with a form. We generally use this control type to store information between client/server exchanges that would otherwise be lost due to the stateless nature of HTTP. The `INPUT` element is used to create a hidden control.

### **object controls**

Authors may insert generic objects in forms such that associated values are submitted along with other controls. Authors create object controls with the OBJECT element.

The elements used to create controls generally appear inside a FORM element, but may also appear outside of a FORM element declaration when they are used to build user interfaces.

### 1.18.3. The <INPUT> tag

The <input> tag specifies an input field where the user can enter data. <input> elements are used within a <form> element to declare input controls that allow users to input data. An input field can vary in many ways, depending on the type attribute.

#### Attributes

| Attribute | Value   | Description  |
|-----------|---|--|
| align     | Left, right, top, middle, bottom  | Not supported in HTML5. Deprecated in HTML 4.01. Specifies the alignment of an image input (only for type="image") |
| alt       | <i>text</i>   | Specifies an alternate text for images (only for type="image")   |
| checked   | checked   | Specifies that an <input> element should be pre-selected when the page loads (for type="checkbox" or type="radio") |
| disabled  | disabled  | Specifies that an <input> element should be disabled   |
| maxlength | <i>number</i>   | Specifies the maximum number of characters allowed in an <input> element   |
| name      | <i>text</i>   | Specifies the name of an <input> element   |
| readonly  | readonly  | Specifies that an input field is read-only   |
| size      | <i>number</i>   | Specifies the width, in characters, of an <input> element  |
| src       | <i>URL</i>  | Specifies the URL of the image to use as a submit button (only for type="image")                                   |
| type      | button, checkbox, color, date, email, file, hidden, image, password, radio, range, reset, search, submit, text, | Specifies the type <input> element to display  |

time, url, week

value                      *text*                      Specifies the value of an `<input>` element

#### 1.18.4. Examples of forms containing INPUT controls

##### Text Fields

`<input type="text" />` defines a one-line input field that a user can enter text into:

`<form>`

First name: `<input type="text" name="firstname" />``<br />`

Last name: `<input type="text" name="lastname" />`

`</form>`

##### Password Field

`<input type="password" />` defines a password field:

`<form>`

Password: `<input type="password" name="pwd" />`

`</form>`

##### Radio Buttons

`<input type="radio" />` defines a radio button. Radio buttons let a user select ONLY ONE of a limited number of choices:

`<form>`

`<input type="radio" name="sex" value="male" />` Male`<br />`

`<input type="radio" name="sex" value="female" />` Female

`</form>`

##### Checkboxes

`<input type="checkbox" />` defines a checkbox. Checkboxes let a user select ZERO or MORE options of a limited number of choices.

`<form>`

`<input type="checkbox" name="vehicle" value="Bike" />` I have a bike`<br />`

`<input type="checkbox" name="vehicle" value="Car" />` I have a car

`</form>`

##### Submit Button

`<input type="submit" />` defines a submit button.

A submit button is used to send form data to a server. The data is sent to the page specified in the form's action attribute. The file defined in the action attribute usually does something with the received input:

`<form name="input" action="html_form_action.asp" method="get">`

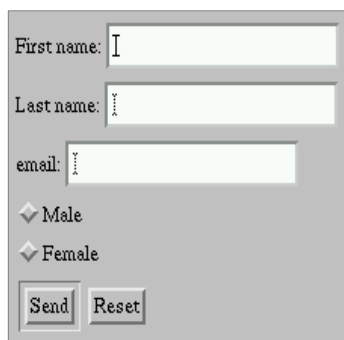
Username: `<input type="text" name="user" />`

```
<input type="submit" value="Submit" />
</form>
```

The following sample HTML fragment defines a simple form that allows the user to enter a first name, last name, email address, and gender. When the submit button is activated, the form will be sent to the program specified by the action attribute.

```
<FORM action="http://somesite.com/prog/adduser" method="post">
  <P>
    First name: <INPUT type="text" name="firstname"><BR>
    Last name: <INPUT type="text" name="lastname"><BR>
    email: <INPUT type="text" name="email"><BR>
    <INPUT type="radio" name="sex" value="Male"> Male<BR>
    <INPUT type="radio" name="sex" value="Female"> Female<BR>
    <INPUT type="submit" value="Send"> <INPUT type="reset">
  </P>
</FORM>
```

This form might be rendered as follows:



In the section on the LABEL element, we discuss marking up labels such as "First name".

### 1.18.5. The BUTTON element

Buttons created with the BUTTON element function just like buttons created with the INPUT element, but they offer richer rendering possibilities: the BUTTON element may have content.

The <button> tag defines a clickable button.

Inside a <button> element you can put content, like text or images. This is the difference between this element and buttons created with the <input> element.

```
<!DOCTYPE html>
<html>
  <body>
    <button type="button" onclick="alert('Hello world!')">Click Me!</button>
  </body>
```

</html>

### 1.18.6. The SELECT and OPTION elements

The <select> element is used to create a drop-down list.

The <option> tags inside the <select> element define the available options in the list.

```
<!DOCTYPE html>
<html>
<body>
<select>
  <option value="volvo">Volvo</option>
  <option value="saab">Saab</option>
  <option value="opel">Opel</option>
  <option value="audi">Audi</option>
</select>
</body>
</html>
```

### 1.19. HTML frames

HTML frames allow authors to present documents in multiple views, which may be independent windows or sub windows. Multiple views offer designers a way to keep certain information visible, while other views are scrolled or replaced. For example, within the same window, one frame might display a static banner, a second a navigation menu, and a third the main document that can be scrolled through or replaced by navigating in the second frame. The <frame> tag defines one particular window (frame) within a <frameset>. Each <frame> in a <frameset> can have different attributes, such as border, scrolling, the ability to resize, etc

#### 1.19.1 The <frameset> tag

The <frameset> tag defines a frameset.

The <frameset> element holds one or more <frame> elements. Each <frame> element can hold a separate document. The <frameset> element specifies HOW MANY columns or rows there will be in the frameset, and HOW MUCH percentage/pixels of space will occupy each of them.

Attributes

| Attribute | Value               | Description  |
|-----------|---------------------|--|
| cols      | <i>Pixels, %, *</i> | Specifies the number and size of columns in a frameset |
| rows      | <i>Pixels, %, *</i> | Specifies the number and size of rows in a frameset    |

#### 1.19.2. The <frame> tag

The <frame> tag defines one particular window (frame) within a <frameset>. Each <frame> in a <frameset> can have different attributes, such as border, scrolling, the ability to resize, etc

#### Attributes

| Attribute    | Value         | Description   |
|--------------|---------------|---|
| frameborder  | 0             | Specifies whether or not to display a border around                         |
|              | 1             | a frame   |
| longdesc     | <i>URL</i>    | Specifies a page that contains a long description of the content of a frame |
| marginheight | <i>pixels</i> | Specifies the top and bottom margins of a frame                             |
| marginwidth  | <i>pixels</i> | Specifies the left and right margins of a frame                             |
| name         | <i>name</i>   | Specifies the name of a frame   |
| noresize     | noresize      | Specifies that a frame is not resizable                                     |
| scrolling    | Yes,<br>auto  | no Specifies whether or not to display scrollbars in a frame                |
| src          | <i>URL</i>    | Specifies the URL of the document to show in a frame                        |

#### Exmple

```
<HTML>
<HEAD>
<TITLE>A simple frameset document</TITLE>
</HEAD>
<FRAMESET cols="20%, 80%">
  <FRAMESET rows="100, 200">
    <FRAME src="contents_of_frame1.html">
    <FRAME src="contents_of_frame2.gif">
  </FRAMESET>
  <FRAME src="contents_of_frame3.html">
</NOFRAMES>
<P>This frameset document contains:
<UL>
  <LI><A href="contents_of_frame1.html">Some neat contents</A>
  <LI><IMG src="contents_of_frame2.gif" alt="A neat image">
  <LI><A href="contents_of_frame3.html">Some other neat contents</A>
</UL>
```

```
</NOFRAMES>
</FRAMESET>
</HTML>
</html>
```

### 1.10. HTML Iframes

An iframe is used to display a web page within a web page. The <iframe> tag specifies an inline frame. An inline frame is used to embed another document within the current HTML document.

Syntax for adding an iframe:

```
<iframe src="URL"></iframe>
```

The URL points to the location of the separate page. The height and width attributes are used to specify the height and width of the iframe. The attribute values are specified in pixels by default, but they can also be in percent (like "80%").

```
<!DOCTYPE html>
<html>
<body>
<iframe src="demo_iframe.htm" width="200" height="200"></iframe>
<p>Some older browsers don't support iframes.</p>
<p>If they don't, the iframe will not be visible.</p>
</body>
</html>
```

### 1.11. HTML <noframes> Tag

The <noframes> tag is a fallback tag for browsers that do not support frames. It can contain all the HTML elements that you can find inside the <body> element of a normal HTML page. The <noframes> element can be used to link to a non-frameset version of the web site or to display a message to users that frames are required. The <noframes> element goes inside the <frameset> element.

```
<!DOCTYPE html>
<html>
<frameset cols="25%,50%,25%">
<frame src="frame_a.htm">
<frame src="frame_b.htm">
<frame src="frame_c.htm">
<noframes>Sorry, your browser does not handle frames!</noframes>
</frameset>
</html>
```



## **Module II**

### **2.1 Introduction to JavaScript**

JavaScript is a computer language specially designed to work with Internet browsers. It lets you create small programs called scripts and embed them inside Hypertext Markup Language (HTML) pages in order to provide interactive content on your Web pages. JScript is Microsoft's implementation of JavaScript. In addition to running within Internet Explorer, Microsoft also provides a version of JScript that can be used as a desktop scripting language with the Windows Script Host.

### **2.2 History**

In 1995 Brendan Eich at Netscape Communications Corporation developed a scripting language called LiveScript, which gave Web page developers greater control over the browser. Later, Sun Microsystems came along and developed a new programming language named Java. Java quickly became a hot item and received an enormous amount of media and industry attention. Netscape added support for Java in Netscape Navigator 2. At the same time, Netscape decided to change the name of LiveScript to JavaScript.

The European Computer Manufacturing Association (ECMA) has taken a lead role in working toward standardizing JavaScript, which it refers to as ECMAScript. The ECMA-262 specification outlines standards with which JavaScript 1.3 is compliant. JavaScript 1.5 is compliant with ECMA-262 revision 3. Like the JavaScript 1.5, JScript 5.6 is based on ECMA-262 revision 3.

### **2.3. FEATURES**

#### **2.3.1. Case sensitivity**

JavaScript is a case-sensitive language. All keywords are in lowercase. All variables, function names, and other identifiers must be typed with a consistent capitalization.

#### **2.3.2. Interpreted language**

JavaScript and JScript are interpreted languages. This means that scripts written in these languages are not compiled before they are executed (as is typical of most programming languages such as C++). Every script statement must first be converted into binary code (a computer language made up of 0s and 1s that the computer can understand) in order to execute. Unlike compiled programs, which are converted to binary code in advance, JavaScript and JScript statements are processed at execution time. This means that they run a little slower than compiled programs.

#### **2.3.3. Object-based scripting language**

This means that they view everything as objects. For JavaScripts, the browser is an object, a window is an object, and a button in a window is an object.

#### **2.3.4. Event-driven programming**

An event is an action that occurs when the user does something such as click on a button or move the pointer over a graphic image. JavaScript enables you to write scripts that are triggered by events.

## **2.4. Client-Side JavaScript**

When a JavaScript interpreter is embedded in a web browser, the result is client-side JavaScript. This is by far the most common variant of JavaScript; when most people refer to JavaScript, they usually mean client-side JavaScript. This book documents client-side JavaScript, along with the core JavaScript language that client-side JavaScript incorporates.

### **2.4.1. JavaScript Is Not Java**

One of the most common misconceptions about JavaScript is that it is a simplified version of Java, the programming language from Sun Microsystems. Other than an incomplete syntactic resemblance and the fact that both Java and JavaScript can provide executable content in web browsers, the two languages are entirely unrelated. The similarity of names is purely a marketing ploy (the language was originally called LiveScript; its name was changed to JavaScript at the last minute).

### **2.4.2. Control Document Appearance and Content**

The JavaScript Document object, through its `write( )` method, which we have already seen, allows you to write arbitrary HTML into a document as the document is being parsed by the browser. For example, you can include the current date and time in a document or display different content on different platforms.

### **2.4.3. Control the Browser**

Several JavaScript objects allow control over the behavior of the browser. The Window object supports methods to pop up dialog boxes to display simple messages to the user and get simple input from the user.

### **2.4.4. Interact with HTML Forms**

Another important aspect of client-side JavaScript is its ability to interact with HTML forms. This capability is provided by the Form object and the form element objects it can contain: Button, Checkbox, Hidden, Password, Radio, Reset, Select, Submit, Text, and Textarea objects. These element objects allow you to read and write the values of the input elements in the forms in a document.

### **2.4.5. Interact with the User**

An important feature of JavaScript is the ability to define event handlers -- arbitrary pieces of code to be executed when a particular event occurs. Usually, these events are initiated by the user, when, for example, she moves the mouse over a hypertext link, enters a value in a form, or clicks the Submit button in a form.

#### **2.4.6. Read and Write Client State with Cookies**

A cookie is a small amount of state data stored permanently or temporarily by the client. Cookies may be transmitted along with a web page by the server to the client, which stores them locally. When the client later requests the same or a related web page, it passes the relevant cookies back to the server, which can use their values to alter the content it sends back to the client. Cookies allow a web page or web site to remember things about the client -- for example, that the user has previously visited the site, has already registered and obtained a password, or has expressed a preference about the color and layout of web pages. Cookies help you provide the state information that is missing from the stateless HTTP protocol of the Web.

When cookies were invented, they were intended for use exclusively by server-side scripts; although stored on the client, they could be read or written only by the server. JavaScript changed this, because JavaScript programs can read and write cookie values and can dynamically generate document content based on the value of cookies.

#### **2.4.7. What JavaScript Can't Do**

Client-side JavaScript has an impressive list of capabilities. Note, however, that they are confined to browser- and document-related tasks. Since client-side JavaScript is used in a limited context, it does not have features that would be required for standalone languages:

- ⤴ JavaScript does not have any graphics capabilities, except for the powerful ability to dynamically generate HTML (including images, tables, frames, forms, fonts, etc.) for the browser to display.
- ⤴ For security reasons, client-side JavaScript does not allow the reading or writing of files. Obviously, you wouldn't want to allow an untrusted program from any random web site to run on your computer and rearrange your files!
- ⤴ JavaScript does not support networking of any kind, except that it can cause the browser to download arbitrary URLs and it can send the contents of HTML forms across the network to server-side scripts and email addresses.

### **2.5. Integrate JavaScript into Your HTML Pages `<script>` tag**

JavaScript are collections of programming statements that you embed in HTML documents by placing them within the `<SCRIPT>` and `</SCRIPT>` tags. There are two places you can put your JavaScript in an HTML page: in either the head or body section. In addition, I have told you that you can either embed JavaScript directly into the HTML page or reference it in as an external .js file. One more way you can integrate JavaScript into an HTML page is as a component in an HTML tag.

#### **2.5.1. `<script>` tag**

This tag is used to embed JavaScript within HTML contents. This tells the browser that the following chunk of text, bounded by the closing `</script>` tag, is not HTML to be displayed, but rather script code to be processed. We call the chunk of code surrounded by the `<script>` and `</script>` tags a script block. The `<script>` tag has a number of attributes, but the most important one for us are the language and type attributes. As we saw earlier, JavaScript is not the only scripting language available, and different scripting languages need to be processed in different ways. We need to tell the browser which scripting language to expect so that it knows how to process it. There are no prizes for guessing that the language attribute, when using JavaScript, takes the value JavaScript. So, our opening script tag will look like this:

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

Including the language attribute is good practice, but within a web page it can be left off. Browsers such as Internet Explorer (IE) and Netscape Navigator (NN) default to a script language of JavaScript. This means that if the browser encounters a `<script>` tag with no language attribute set, it assumes that the script block is written in JavaScript. However, it is good practice to always include the language attribute.

However, the web standards have depreciated the requirements for the language attribute; this means the final aim is for its removal. In its place we have the type attribute, and this is made mandatory by the web standards, although most current browsers will let us get away without including it. In the case of script, we need to specify that the type is text/JavaScript, as follows:

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

However, older browsers do not support the type attribute, so for compatibility it's best to include both, as shown here:

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

We cannot write markup tags directly within script tag instead it has to be included directly within output string.

### **2.5.2. Placing JavaScript in the Body Section of the HTML page**

JavaScript embedded with the `<SCRIPT>` and `</SCRIPT>` tags can be placed anywhere in the body section of an HTML page. Scripts embedded in the body section are executed as part of the HTML document when the page loads. This enables the script to begin executing automatically as the page loads. For example, the statements shown below demonstrate how to embed a JavaScript within the body section of an HTML page.

```
<BODY>
```

```
<SCRIPT LANGUAGE="JavaScript" TYPE="Text/JavaScript">
```

```
document.write("This JavaScript is located in the body section");
```

```
</SCRIPT>
```

</BODY>

Similarly, you can embed multiple JavaScripts in HTML pages:

<BODY>

<SCRIPT LANGUAGE="JavaScript" TYPE="Text/JavaScript">

*document.write("This first JavaScript is located in the body section");*

</SCRIPT>

<BR>

<SCRIPT LANGUAGE="JavaScript" TYPE="Text/JavaScript">

*document.write("This second JavaScript is also located in the body section");*

</SCRIPT>

</BODY>

### 2.5.3. Placing JavaScript in the Head Section of the HTML page

JavaScript also can be placed anywhere within the head section of your HTML pages. Unlike scripts embedded within the body section of HTML pages, scripts embedded in the head section are not necessarily automatically executed when the page loads. In some cases, they are executed only when called for execution by other statements within the HTML page. Most JavaScript programmers move all functions and most variables to the head section because this ensures that they will be defined before being referenced by scripts located in the body section of the page.

The following statements show an HTML page with a JavaScript embedded in the head section. This script will automatically execute when the HTML page is loaded.

<HTML>

<HEAD>

<TITLE>Script 1.3 - Sample HTML Page</TITLE>

<SCRIPT LANGUAGE="JavaScript" TYPE="Text/JavaScript">

*window.alert("This JavaScript is located in the head section");*

</SCRIPT>

</HEAD>

<BODY>

</BODY>

<HTML>

This next HTML page contains an embedded JavaScript that will not automatically execute when the HTML is loaded by the browser.

<HTML>

<HEAD>

<TITLE>Script 1.4 - Sample HTML Page</TITLE>

```

<SCRIPT LANGUAGE="JavaScript" TYPE="Text/JavaScript">
    function DisplayMsg() {
        window.alert("This JavaScript is located in the head section");
    }
</SCRIPT>
</HEAD>
<BODY onLoad="DisplayMsg()">
</BODY>

```

Just understand that they can be used to create JavaScripts in the HTML page's head section that do not execute automatically.

#### 2.5.4. Referencing JavaScript in an External .js File

To store your JavaScript in external files, you need to save them as plain text files with a .js file extension. You then need to add the SRC attribute to the opening <SCRIPT> tag in your HTML page as demonstrated here.

Create a file Test.js

```

document.write("This is an external JavaScript.");
function DisplayMsg() {
    window.alert("This JavaScript is located in the head section");
}
window.alert("This JavaScript is located in the head section");

```

Create an HTML file to link the above .js file

```

<HTML>
<HEAD>
<SCRIPT SRC=" Test.js " LANGUAGE="JavaScript" TYPE="Text/JavaScript">
</SCRIPT>
<TITLE>Script 1.3 - Sample HTML Page</TITLE>
</HEAD>
<BODY onload="DisplayMsg()">
</BODY>
</HTML>

```

In this example, an external JavaScript named Test.js has been specified. This external JavaScript can contain any number of JavaScript statements. However, it cannot contain any HTML whatsoever. Otherwise you'll end up with an error. For example, the contents of the Test.js script might be as simple as this:

```

document.write("This is an external JavaScript.");

```

### 2.5.5. Placing JavaScript in an HTML Tag

JavaScript can also be placed within HTML tags, as shown in the following example.

```
<BODY onLoad=document.write("Hello World!")> </BODY>
```

In this example, the JavaScript `onLoad=document.write("Hello World!")` statement has been added to the HTML `<BODY>` tag. This particular JavaScript statement tells the browser to write the enclosed text when the browser first loads the HTML page.

### 2.5.6. JavaScript Pseudo-URL

In most JavaScript-aware browsers, it is possible to invoke a script using the JavaScript pseudo-URL. A pseudo-URL like **javascript: alert('hello')** would invoke a simple alert displaying “hello” when typed directly in the browser’s address bar, as shown here:

```
<a href="javascript: alert('hello I am a pseudo-URL script');">
```

```
Click to invoke</a>
```

Type the below code at the browsers address bar

```
javascript:x = 3; (x < 5)? "x is less": "x is greater"
```

```
javascript:d = new Date( ); typeof d;
```

## 2.6 Character Set

JavaScript programs are written using the Unicode character set. Unlike the 7-bit ASCII encoding, which is useful only for English, and the 8-bit ISO Latin-1 encoding, which is useful only for English and major Western European languages, the 16-bit Unicode encoding can represent virtually every written language in common use on the planet. This is an important feature for internationalization and is particularly important for programmers who do not speak English.

## 2.7 Token

The smallest indivisible lexical unit of the language. A contiguous sequence of characters whose meaning would change if separated by a space.

### 2.7.1. Reserved Words

There are a number of reserved words in JavaScript. These are words that you cannot use as identifiers (variable names, function names, and loop labels) in your JavaScript programs.

### 2.7.2 Identifiers

An identifier is simply a name. In JavaScript, identifiers are used to name variables and functions and to provide labels for certain loops in JavaScript code.

The rules for legal identifier names are

- The first character must be a letter, an underscore (`_`), or a dollar sign (`$`).
- Subsequent characters may be any letter or digit or an underscore or dollar sign.
- Numbers are not allowed as the first character
- Keywords cannot be used as identifiers.

Eg: i, my\_variable\_name, v13, \_dummy, \$str

### 2.7.3. Literals

A literal is a data value that appears directly in a program. The following are all literals:

```
12          // The number twelve
1.2         // The number one point two
"hello world" // A string of text
'Hi'        // Another string
true        // A boolean value
false       // The other boolean value
/javascript/gi // A "regular expression" literal (for pattern matching)
null        // Absence of an object
{ x:1, y:2 } // An object initializer
[1,2,3,4,5]  // An array initializer
```

#### Integer Literals

In a JavaScript program, a base-10 integer is written as a sequence of digits.

For example: 0, 3, 10000000

The JavaScript number format allows you to exactly represent all integers between -9007199254740992 ( $-2^{53}$ ) and 9007199254740992 ( $2^{53}$ ), inclusive. If you use integer values larger than this, you may lose precision in the trailing digits.

A hexadecimal literal begins with "0x" or "0X", followed by a string of hexadecimal digits.

Examples of hexadecimal integer literals are: 0xff //  $15 \times 16 + 15 = 255$  (base 10), 0xCAFE911

An octal literal begins with the digit 0 and is followed by a sequence of digits.

For example: 0377 //  $3 \times 64 + 7 \times 8 + 7 = 255$  (base 10)

Since some implementations support octal literals and some do not, you should never write an integer literal with a leading zero -- you cannot know whether an implementation will interpret it as an octal or decimal value.

#### Floating-Point Literals

Floating-point literals can have a decimal point; they use the traditional syntax for real numbers. A real value is represented as the integral part of the number, followed by a decimal point and the fractional part of the number.

For example: 3.14, 2345.789, .333333333333333333

Floating-point literals may also be represented using exponential notation: a real number followed by the letter e (or E), followed by an optional plus or minus sign, followed by an integer exponent. This notation represents the real number multiplied by 10 to the power of the exponent.

The syntax is:



*[digits][.digits][(E/e)[(+/-)]digits]*

For example: 6.02e23     // 6.02 x 10<sup>23</sup>, 1.4738223E-32 // 1.4738223 x 10<sup>-32</sup>

## String Literals

A string is a sequence of zero or more Unicode characters enclosed within single or double quotes (' or "). Double-quote characters may be contained within strings delimited by single-quote characters, and single-quote characters may be contained within strings delimited by double quotes. String literals must be written on a single line; they may not be broken across two lines. If you need to include a newline character in a string literal, use the character sequence \n, which is documented in the next section.

Examples of string literals are: ""    // The empty string: it has zero characters, 'testing', "3.14", 'name="myform"'

"Wouldn't you prefer O'Reilly's book?", "This string\nhas two lines", "pi is the ratio of a circle's circumference to its diameter"

## Boolean Literals

There are two Boolean literals true and false.

## Function Literals

A function literal is defined with the function keyword, followed by an optional function name, followed by a parenthesized list of function arguments and the body of the function within curly braces. In other words, a function literal looks just like a function definition, except that it does not have to have a name. The big difference is that function literals can appear within other JavaScript expressions. Thus, instead of defining the function square( ) with a function definition:

```
function square(x) { return x*x; }
```

We can define it with a function literal:

```
var square = function(x) { return x*x; }
```

There is one other way to define a function: you can pass the argument list and the body of the function as strings to the Function( ) constructor. For example:

```
var square = new Function("x", "return x*x;");
```

Defining a function in this way is not often useful. It is usually awkward to express the function body as a string, and in many JavaScript implementations, functions defined in this way will be less efficient than functions defined in either of the other two ways.

## Object Literals

An object literal (also called an object initializer) consists of a comma-separated list of colon-separated property/value pairs, all enclosed within curly braces. Thus, the object point in the previous code could also be created and initialized with this line:

```
var point = { x:2.3, y:-1.2 };
```

Object literals can also be nested. For example:

```
var rectangle = { upperLeft: { x: 2, y: 2 }, lowerRight: { x: 4, y: 4 } };
```

Finally, the property values used in object literals need not be constant -- they can be arbitrary JavaScript expressions:

```
var square = { upperLeft: { x:point.x, y:point.y }, lowerRight: { x:(point.x + side), y:(point.y+side) } };
```

### **Array Literals**

An array literal (or array initializer) is a comma-separated list of values contained within square brackets. The values within the brackets are assigned sequentially to array indexes starting with zero.

```
var a = [1.2, "JavaScript", true, { x:1, y:3 }];
```

Like object literals, array literals can be nested:

```
var matrix = [[1,2,3], [4,5,6], [7,8,9]];
```

Also, as with object literals, the elements in array literals can be arbitrary expressions and need not be restricted to constants:

```
var base = 1024;
```

```
var table = [base, base+1, base+2, base+3];
```

Undefined elements can be included in an array literal by simply omitting a value between commas. For example, the following array contains five elements, including three undefined elements:

```
var sparseArray = [1,,5];
```

### **2.7.4. Operators**

Operators can be categorized based on the number of operands they expect. Most JavaScript operators, like the + operator we saw earlier, are binary operators that combine two expressions into a single, more complex expression. That is, they operate on two operands. JavaScript also supports a number of unary operators, which convert a single expression into a single, more complex expression.

#### **Arithmetic Operators**

Having explained operator precedence, associativity, and other background material, we can start to discuss the operators themselves. This section details the arithmetic operators:

##### **Addition (+)**

The + operator adds numeric operands or concatenates string operands. If one operand is a string, the other is converted to a string and the two strings are then concatenated. Object operands are converted to numbers or strings that can be added or concatenated. The conversion is performed by the `valueOf()` method and/or the `toString()` method of the object.

##### **Subtraction (-)**

When `-` is used as a binary operator, it subtracts its second operand from its first operand. If used with non-numeric operands, it attempts to convert them to numbers.

### **Multiplication (\*)**

The `*` operator multiplies its two operands. If used with non-numeric operands, it attempts to convert them to numbers.

### **Division (/)**

The `/` operator divides its first operand by its second. If used with non-numeric operands, it attempts to convert them to numbers. If you are used to programming languages that distinguish between integer and floating-point numbers, you might expect to get an integer result when you divide one integer by another. In JavaScript, however, all numbers are floating-point, so all divisions have floating-point results: `5/2` evaluates to `2.5`, not `2`. Division by zero yields positive or negative infinity, while `0/0` evaluates to `NaN`.

### **Modulo (%)**

The `%` operator computes the first operand modulo the second operand. That is, it returns the remainder when the first operand is divided by the second operand an integral number of times. If used with non-numeric operands, the modulo operator attempts to convert them to numbers. The sign of the result is the same as the sign of the first operand. For example, `5 % 2` evaluates to `1`.

While the modulo operator is typically used with integer operands, it also works for floating-point values. For example, `-4.3 % 2.1` evaluates to `-0.1`.

### **Unary minus (-)**

When `-` is used as a unary operator, before a single operand, it performs unary negation. In other words, it converts a positive value to an equivalently negative value, and vice versa. If the operand is not a number, this operator attempts to convert it to one.

### **Unary plus (+)**

For symmetry with the unary minus operator, JavaScript also has a unary plus operator. This operator allows you to explicitly specify the sign of numeric literals, if you feel that this will make your code clearer:

```
var profit = +1000000;
```

In code like this, the `+` operator does nothing; it simply evaluates to the value of its argument. Note, however, that for non-numeric arguments, the `+` operator has the effect of converting the argument to a number. It returns `NaN` if the argument cannot be converted.

### **Increment (++)**

The `++` operator increments (i.e., adds 1 to) its single operand, which must be a variable, an element of an array, or a property of an object. If the value of this variable, element, or property is not a number, the operator first attempts to convert it to one. The precise behavior of this operator

depends on its position relative to the operand. When used before the operand, where it is known as the pre-increment operator, it increments the operand and evaluates to the incremented value of that operand. When used after the operand, where it is known as the post-increment operator, it increments its operand but evaluates to the incremented value of that operand. If the value to be incremented is not a number, it is converted to one by this process.

For example, the following code sets both *i* and *j* to 2:

```
i = 1;
```

```
j = ++i;
```

But these lines set *i* to 2 and *j* to 1:

```
i = 1;
```

```
j = i++;
```

### **Decrement (--)**

The -- operator decrements (i.e., subtracts 1 from) its single numeric operand, which must be a variable, an element of an array, or a property of an object. If the value of this variable, element, or property is not a number, the operator first attempts to convert it to one. Like the ++ operator, the precise behavior of -- depends on its position relative to the operand. When used before the operand, it decrements and returns the decremented value. When used after the operand, it decrements the operand but returns the undecrement value.

### **Equality Operators**

This section describes the JavaScript equality and inequality operators. These are operators that compare two values to determine whether they are the same or different and return a boolean value (true or false) depending on the result of the comparison. They are most commonly used in things like if statements and for loops, to control the flow of program execution.

#### **Equality (==) and Identity (===)**

The == and === operators check whether two values are the same, using two different definitions of sameness. Both operators accept operands of any type, and both return true if their operands are the same and false if they are different. The === operator is known as the identity operator, and it checks whether its two operands are "identical" using a strict definition of sameness. The == operator is known as the equality operator; it checks whether its two operands are "equal" using a more relaxed definition of sameness that allows type conversions.

#### **Inequality (!=) and Non-Identity (!==)**

The != and !== operators test for the exact opposite of the == and === operators. The != inequality operator returns false if two values are equal to each other and returns true otherwise. The !== non-identity operator returns false if two values are identical to each other and returns true otherwise.

## Relational Operators

This section describes the JavaScript relational operators. These are operators that test for a relationship (such as "less-than" or "property-of") between two values and return true or false depending on whether that relationship exists.

### Comparison Operators

The most commonly used types of relational operators are the comparison operators, which are used to determine the relative order of two values. The comparison operators are:

Less than (<)

The < operator evaluates to true if its first operand is less than its second operand; otherwise it evaluates to false.

Greater than (>)

The > operator evaluates to true if its first operand is greater than its second operand; otherwise it evaluates to false.

Less than or equal (<=)

The <= operator evaluates to true if its first operand is less than or equal to its second operand; otherwise it evaluates to false.

Greater than or equal (>=)

The >= operator evaluates to true if its first operand is greater than or equal to its second operand; otherwise it evaluates to false.

### The in Operator

The in operator expects a left hand operand that is or can be converted to a string. It expects a right hand operand that is an object (or array). It evaluates to true if the left hand value is the name of a property of the right hand object. For example:

```
var point = { x:1, y:1 };    // Define an object
var has_x_coord = "x" in point; // Evaluates to true
var has_y_coord = "y" in point; // Evaluates to true
var has_z_coord = "z" in point; // Evaluates to false; not a 3-D point
var ts = "toString" in point; // Inherited property; evaluates to true
```

### The instanceof Operator

The instanceof operator expects a left hand operand that is an object and a right hand operand that is the name of a class of objects. The operator evaluates to true if the left hand object is an instance of the right hand class and evaluates to false otherwise. Thus, the right hand operand of instanceof should be the name of a constructor function. Note that all objects are instances of Object. For example:

```
var d = new Date(); // Create a new object with the Date( ) constructor
```

```

d instanceof Date; // Evaluates to true; d was created with Date( )
d instanceof Object; // Evaluates to true; all objects are instances of Object
d instanceof Number; // Evaluates to false; d is not a Number object
var a = [1, 2, 3]; // Create an array with array literal syntax
a instanceof Array; // Evaluates to true; a is an array
a instanceof Object; // Evaluates to true; all arrays are objects
a instanceof RegExp; // Evaluates to false; arrays are not regular expressions

```

If the left hand operand of instanceof is not an object, or if the right hand operand is an object that is not a constructor function, instanceof returns false. On the other hand, it returns a runtime error if the right hand operand is not an object at all.

## String Operators

As we've discussed in the previous sections, there are several operators that have special effects when their operands are strings.

The + operator concatenates two string operands. That is, it creates a new string that consists of the first string followed by the second. For example, the following expression evaluates to the string "hello there":

```
"hello" + " " + "there"
```

*And the following lines produce the string "22":*

```
a = "2"; b = "2";
```

```
c = a + b;
```

The < , <=, >, and >= operators compare two strings to determine what order they fall in. The comparison uses alphabetical order. As noted above, however, this alphabetical order is based on the Unicode character encoding used by JavaScript. In this encoding, all capital letters in the Latin alphabet come before (are less than) all lowercase letters, which can cause unexpected results.

The == and != operators work on strings, but, as we've seen, these operators work for all data types, and they do not have any special behavior when used with strings.

The + operator is a special one -- it gives priority to string operands over numeric operands. As noted earlier, if either operand to + is a string (or an object), the other operand is converted to a string (or both operands are converted to strings) and concatenated, rather than added. On the other hand, the comparison operators perform string comparison only if both operands are strings. If only one operand is a string, JavaScript attempts to convert it to a number. The following lines illustrate:

```
1 + 2 // Addition. Result is 3.
```

```
"1" + "2" // Concatenation. Result is "12".
```

```
"1" + 2 // Concatenation; 2 is converted to "2". Result is "12".
```

```
11 < 3 // Numeric comparison. Result is false.
```

*"11" < "3" // String comparison. Result is true.*

*"11" < 3 // Numeric comparison; "11" converted to 11. Result is false.*

*"one" < 3 // Numeric comparison; "one" converted to NaN. Result is false.*

*// In JavaScript 1.1, this causes an error instead of NaN.*

Finally, it is important to note that when the + operator is used with strings and numbers, it may not be associative. That is, the result may depend on the order in which operations are performed. This can be seen with examples like these:

*s = 1 + 2 + " blind mice"; // Yields "3 blind mice"*

*t = "blind mice: " + 1 + 2; // Yields "blind mice: 12"*

The reason for this surprising difference in behavior is that the + operator works from left to right, unless parentheses change this order. Thus, the last two examples are equivalent to these:

*s = (1 + 2) + "blind mice"; // 1st + yields number; 2nd yields string*

*t = ("blind mice: " + 1) + 2; // Both operations yield strings*

## **Logical Operators**

The logical operators are typically used to perform Boolean algebra. They are often used in conjunction with comparison operators to express complex comparisons that involve more than one variable and are frequently used with the if, while, and for statements.

### **Logical AND (&&)**

When used with boolean operands, the && operator performs the Boolean AND operation on the two values: it returns true if and only if both its first operand and its second operand are true. If one or both of these operands is false, it returns false.

The actual behavior of this operator is somewhat more complicated. It starts by evaluating its first operand, the expression on its left. If the value of this expression can be converted to false (for example, if the left operand evaluates to null, 0, "", or undefined), the operator returns the value of the left hand expression. Otherwise, it evaluates its second operand, the expression on its right, and returns the value of that expression.

### **Logical OR (||)**

When used with boolean operands, the || operator performs the Boolean OR operation on the two values: it returns true if either the first operand or the second operand is true, or if both are true. If both operands are false, it returns false. It starts by evaluating its first operand, the expression on its left. If the value of this expression can be converted to true, it returns the value of the left hand expression. Otherwise, it evaluates its second operand, the expression on its right, and returns the value of that expression.

## **Logical NOT (!)**

The ! Operator is a unary operator; it is placed before a single operand. Its purpose is to invert the boolean value of its operand. For example, if the variable a has the value true (or is a value that converts to true), !a has the value false. And if the expression p && q evaluates to false (or to a value that converts to false), !(p && q) evaluates to true. Note that you can convert any value x to a boolean value by applying this operator twice: !!x.

## **Bitwise Operators**

Despite the fact that all numbers in JavaScript are floating-point, the bitwise operators require numeric operands that have integer values. They operate on these integer operands using a 32-bit integer representation instead of the equivalent floating-point representation. Four of these operators perform Boolean algebra on the individual bits of the operands, behaving as if each bit in each operand were a boolean value and performing similar operations to those performed by the logical operators we saw earlier. The other three bitwise operators are used to shift bits left and right.

The bitwise operators are:

### **Bitwise AND (&)**

The & operator performs a Boolean AND operation on each bit of its integer arguments. A bit is set in the result only if the corresponding bit is set in both operands. For example, 0x1234 & 0x00FF evaluates to 0x0034.

### **Bitwise OR (|)**

The | operator performs a Boolean OR operation on each bit of its integer arguments. A bit is set in the result if the corresponding bit is set in one or both of the operands. For example, 9 | 10 evaluates to 11.

### **Bitwise XOR (^)**

The ^ operator performs a Boolean exclusive OR operation on each bit of its integer arguments. Exclusive OR means that either operand one is true or operand two is true, but not both. A bit is set in this operation's result if a corresponding bit is set in one (but not both) of the two operands. For example, 9 ^ 10 evaluates to 3.

### **Bitwise NOT (~)**

The ~ operator is a unary operator that appears before its single integer argument. It operates by reversing all bits in the operand. Because of the way signed integers are represented in JavaScript, applying the ~ operator to a value is equivalent to changing its sign and subtracting 1. For example ~0x0f evaluates to 0xfffff0, or -16.

### **Shift left (<<)**

The << operator moves all bits in its first operand to the left by the number of places specified in the second operand, which should be an integer between 0 and 31. For example, in the operation a



`<< 1`, the first bit (the ones bit) of a becomes the second bit (the twos bit), the second bit of a becomes the third, etc. A zero is used for the new first bit, and the value of the 32nd bit is lost. Shifting a value left by one position is equivalent to multiplying by 2, shifting two positions is equivalent to multiplying by 4, etc. For example, `7 << 1` evaluates to 14.

### **Shift right with sign (>>)**

The `>>` operator moves all bits in its first operand to the right by the number of places specified in the second operand (an integer between 0 and 31). Bits that are shifted off the right are lost. The bits filled in on the left depend on the sign bit of the original operand, in order to preserve the sign of the result. If the first operand is positive, the result has zeros placed in the high bits; if the first operand is negative, the result has ones placed in the high bits. Shifting a value right one place is equivalent to dividing by 2 (discarding the remainder), shifting right two places is equivalent to integer division by 4, and so on. For example, `7 >> 1` evaluates to 3 and `-7 >> 1` evaluates to -4.

### **Shift right with zero fill (>>>)**

The `>>>` operator is just like the `>>` operator, except that the bits shifted in on the left are always zero, regardless of the sign of the first operand. For example, `-1 >> 4` evaluates to -1, but `-1 >>> 4` evaluates to 268435455 (0x0ffffff).

## **Assignment Operators**

The `=` operator is used in JavaScript to assign a value to a variable. For example:

```
i = 0
```

You can write code like this to assign a single value to multiple variables:

```
i = j = k = 0;
```

### **Assignment with Operation**

Besides the normal `=` assignment operator, JavaScript supports a number of other assignment operators that provide shortcuts by combining assignment with some other operation. For example, the `+=` operator performs addition and assignment. The following expression:

```
total += sales_tax
```

*is equivalent to this one:*

```
total = total + sales_tax
```

Similar operators include `-=`, `*=`, `&=`, and so on.

### **The Conditional Operator (?:)**

The conditional operator is the only ternary operator (three operands) in JavaScript and is sometimes actually called the ternary operator. This operator is sometimes written `?:`, although it does not appear quite that way in code. Because this operator has three operands, the first goes before the `?`, the second goes between the `?` and the `:`, and the third goes after the `:`. It is used like this:

*$x > 0 ? x*y : -x*y$*

*greeting = "hello " + (username != null ? username : "there");*

### **The typeof Operator**

typeof is a unary operator that is placed before its single operand, which can be of any type. Its value is a string indicating the data type of the operand.

The typeof operator evaluates to "number", "string", or "boolean" if its operand is a number, string, or boolean value. It evaluates to "object" for objects, arrays, and (surprisingly) null. It evaluates to "function" for function operands and to "undefined" if the operand is undefined.

*typeof i*

*(typeof value == "string") ? "" + value + "" : value*

### **The Object Creation Operator (new)**

The new operator creates a new object and invokes a constructor function to initialize it. new is a unary operator that appears before a constructor invocation. It has the following syntax:

*new constructor(arguments)*

*o = new Object; // Optional parentheses omitted here*

*d = new Date( ); // Returns a Date object representing the current time*

*c = new Rectangle(3.0, 4.0, 1.5, 2.75); // Create an object of class Rectangle*

*obj[i] = new constructors[i]( );*

### **The delete Operator**

delete is a unary operator that attempts to delete the object property, array element, or variable specified as its operand. It returns true if the deletion was successful, and false if the operand could not be deleted. Not all variables and properties can be deleted: some built-in core and client-side properties are immune from deletion, and user-defined variables declared with the var statement cannot be deleted. If delete is invoked on a nonexistent property, it returns true.

If you are a C++ programmer, note that the delete operator in JavaScript is nothing like the delete operator in C++. In JavaScript, memory deallocation is handled automatically by garbage collection, and you never have to worry about explicitly freeing up memory. Thus, there is no need for a C++-style delete to delete entire objects.

*var o = {x:1, y:2}; // Define a variable; initialize it to an object*

*delete o.x; // Delete one of the object properties; returns true*

*typeof o.x; // Property does not exist; returns "undefined"*

*delete o.x; // Delete a nonexistent property; returns true*

*delete o; // Can't delete a declared variable; returns false*

*delete 1; // Can't delete an integer; returns true*

*x = 1; // Implicitly declare a variable without var keyword*

```
delete x;           // Can delete this kind of variable; returns true
x;                  // Runtime error: x is not defined
```

### 2.7.5. Comments

JavaScript, like Java, supports both C++ and C-style comments. Any text between a `//` and the end of a line is treated as a comment and is ignored by JavaScript. Any text between the characters `/*` and `*/` is also treated as a comment. These C-style comments may span multiple lines but may not be nested. The following lines of code are all legal JavaScript comments:

```
// This is a single-line comment.
/* This is also a comment */ // and here is another comment.
```

## 2.8 Data Types and Values

Computer programs work by manipulating values, such as the number 3.14 or the text "Hello World". The types of values that can be represented and manipulated in a programming language are known as data types, and one of the most fundamental characteristics of a programming language is the set of data types it supports. JavaScript allows you to work with three **primitive data types: numbers, strings of text (known as "strings"), and boolean** truth values (known as "booleans"). JavaScript also defines two **trivial data types, null and undefined**, each of which defines only a single value.

In addition to these primitive data types, JavaScript supports a composite data type known as object. An object (that is, a member of the data type object) represents a collection of values (either primitive values, like numbers and strings, or composite values, like other objects). Objects in JavaScript have a dual nature: an object can represent an unordered collection of named values or an ordered collection of numbered values. In the latter case, the object is called an array. Although objects and arrays are fundamentally the same data type in JavaScript, they behave quite differently and will usually be considered distinct types throughout this book.

JavaScript defines another special kind of object, known as a function. A function is an object that has executable code associated with it.

In addition to functions and arrays, core JavaScript defines a few other specialized kinds of objects. These objects do not represent new data types, just new classes of objects. The Date class defines objects that represent dates, the RegExp class defines objects that represent regular expressions (a powerful pattern-matching), and the Error class defines objects that represent syntax and runtime errors that can occur in a JavaScript program.

### 2.8.1. Numbers

Numbers are the most basic data type; they require very little explanation. JavaScript differs from programming languages such as C and Java in that it does not make a distinction between integer values and floating-point values. All numbers in JavaScript are represented as floating-point values.

JavaScript represents numbers using the 64-bit floating-point format defined by the IEEE 754 standard, which means it can represent numbers as large as  $\pm 1.7976931348623157 \times 10^{308}$  and as small as  $\pm 5 \times 10^{-324}$ .

### Special Numeric Values

JavaScript uses several special numeric values. When a floating-point value becomes larger than the largest representable finite number, the result is a special infinity value, which JavaScript prints as `Infinity`. Similarly, when a negative value becomes lower than the last representable negative number, the result is negative infinity, printed as `-Infinity`.

Another special JavaScript numeric value is returned when a mathematical operation (such as division of zero by zero) yields an undefined result or an error. In this case, the result is the special not-a-number value, printed as `NaN`. The not-a-number value behaves unusually: it does not compare equal to any number, including itself! For this reason, a special function, `isNaN()`, is required to test for this value. A related function, `isFinite()`, tests whether a number is not `NaN` and is not positive or negative infinity.

### 2.8.2. Strings

A string is a sequence of Unicode letters, digits, punctuation characters, and so on - it is the JavaScript data type for representing text. As you'll see shortly, you can include string literals in your programs by enclosing them in matching pairs of single or double quotation marks. Note that JavaScript does not have a character data type such as `char`, like C, C++, and Java do. To represent a single character, you simply use a string that has a length of 1.

### 2.8.3. Boolean Values

The number and string data types have a large or infinite number of possible values. The boolean data type, on the other hand, has only two. The two legal boolean values are represented by the literals `true` and `false`. A boolean value represents a truth value -- it says whether something is true or not.

### 2.8.4. Functions

A function is a piece of executable code that is defined by a JavaScript program or predefined by the JavaScript implementation. Although a function is defined only once, a JavaScript program can execute or invoke it any number of times.

```
function square(x) // The function is named square. It expects one argument, x.
{
    // The body of the function begins here.
    return x*x;      // The function squares its argument and returns that value.
}
// The function ends here.
```

### 2.8.5. Objects

An object is a collection of named values. These named values are usually referred to as properties of the object. (Sometimes they are called fields of the object, but this usage can be confusing.) To refer to a property of an object, you refer to the object, followed by a period and the name of the property. For example, if an object named `image` has properties named `width` and `height`, we can refer to those properties like this:

*`image.width`*

*`image.height`*

Properties of objects are, in many ways, just like JavaScript variables; they can contain any type of data, including arrays, functions, and other objects. Thus, you might see JavaScript code like this:

*`document.myform.button`*

This code refers to the `button` property of an object that is itself stored in the `myform` property of an object named `document`.

As mentioned earlier, when a function value is stored in a property of an object, that function is often called a method, and the property name becomes the method name. To invoke a method of an object, use the `.` syntax to extract the function value from the object, and then use the `( )` syntax to invoke that function. For example, to invoke the `write( )` method of the `Document` object, you can use code like this:

*`document.write("this is a test");`*

Objects in JavaScript have the ability to serve as associative arrays -- that is, they can associate arbitrary data values with arbitrary strings. When an object is used in this way, a different syntax is generally required to access the object's properties: a string containing the name of the desired property is enclosed within square brackets. Using this syntax, we could access the properties of the `image` object mentioned previously with code like this:

*`image["width"]`*

*`image["height"]`*

Associative arrays are a powerful data type; they are useful for a number of programming techniques.

### Creating Objects

Objects are created by invoking special constructor functions. For example, the following lines all create new objects:

*`var o = new Object( );`*

*`var now = new Date( );`*

*`var pattern = new RegExp("\\sjava\\s", "i");`*

Once you have created an object of your own, you can use and set its properties however you desire:

```
var point = new Object( );  
point.x = 2.3;  
point.y = -1.2;
```

### 2.8.6. Arrays

An array is a collection of data values, just as an object is. While each data value contained in an object has a name, each data value in an array has a number, or index. In JavaScript, you retrieve a value from an array by enclosing an index within square brackets after the array name. For example, if an array is named *a*, and *i* is a non-negative integer, *a[i]* is an element of the array. Array indexes begin with zero. Thus, *a[2]* refers to the third element of the array *a*.

Arrays may contain any type of JavaScript data, including references to other arrays or to objects or functions. For example:

```
document.images[1].width
```

This code refers to the *width* property of an object stored in the second element of an array stored in the *images* property of the *document* object.

### Creating Arrays

Arrays can be created with the *Array( )* constructor function. Once created, any number of indexed elements can easily be assigned to the array:

```
var a = new Array( );  
a[0] = 1.2;  
a[1] = "JavaScript";  
a[2] = true;  
a[3] = { x:1, y:3 };
```

Arrays can also be initialized by passing array elements to the *Array( )* constructor. Thus, the previous array-creation and -initialization code could also be written:

```
var a = new Array(1.2, "JavaScript", true, { x:1, y:3 });
```

If you pass only a single number to the *Array( )* constructor, it specifies the length of the array. Thus:

```
var a = new Array(10); //creates a new array with 10 undefined elements.
```

### 2.8.7. null

The JavaScript keyword *null* is a special value that indicates no value. *null* is usually considered to be a special value of object type -- a value that represents no object. *null* is a unique value, distinct from all other values. When a variable holds the value *null*, you know that it does not contain a valid object, array, number, string, or boolean value.

### 2.8.8. undefined

Another special value used occasionally by JavaScript is the undefined value returned when you use either a variable that has been declared but never had a value assigned to it, or an object property that does not exist. Note that this special undefined value is not the same as null.

Although null and the undefined value are distinct, the == equality operator considers them to be equal to one another. Consider the following:

```
my.prop == null
```

This comparison is true either if the my.prop property does not exist or if it does exist but contains the value null. Since both null and the undefined value indicate an absence of value, this equality is often what we want. However, if you truly must distinguish between a null value and an undefined value, use the === identity operator or the typeof operator.

Unlike null, undefined is not a reserved word in JavaScript.

```
var undefined;
```

By declaring but not initializing the variable, you assure that it has the undefined value.

### 2.8.9. The Date Object

The previous sections have described all of the fundamental data types supported by JavaScript. Date and time values are not one of these fundamental types, but JavaScript does provide a class of object that represents dates and times and can be used to manipulate this type of data. A Date object in JavaScript is created with the new operator and the Date( ) constructor.

```
var now = new Date( ); // Create an object holding the current date and time.
```

```
// Create a Date object representing Christmas.
```

```
// Note that months are zero-based, so December is month 11!
```

```
var xmas = new Date(2000, 11, 25);
```

Methods of the Date object allow you to get and set the various date and time values and to convert the Date to a string, using either local time or GMT time. For example:

```
xmas.setFullYear(xmas.getFullYear( ) + 1); // Change the date to next Christmas.
```

```
var weekday = xmas.getDay( ); // Christmas falls on a Tuesday in 2001.
```

```
document.write("Today is: " + now.toLocaleString( )); // Current date/time.
```

The Date object also defines functions (not methods; they are not invoked through a Date object) to convert a date specified in string or numeric form to an internal millisecond representation that is useful for some kinds of date arithmetic.

### 2.8.10. Regular Expressions

Regular expressions provide a rich and powerful syntax for describing textual patterns; they are used for pattern matching and for implementing search and replace operations. JavaScript has adopted the Perl programming language syntax for expressing regular expressions

Regular expressions are represented in JavaScript by the `RegExp` object and may be created using the `RegExp()` constructor. Like the `Date` object, the `RegExp` object is not one of the fundamental data types of JavaScript; it is simply a specialized kind of object provided by all conforming JavaScript implementations.

Unlike the `Date` object, however, `RegExp` objects have a literal syntax and can be encoded directly into JavaScript 1.2 programs. Text between a pair of slashes constitutes a regular expression literal. The second slash in the pair can also be followed by one or more letters, which modify the meaning of the pattern. For example:

```
/^HTML/  
/[1-9][0-9]*/  
\bjavascript\b/i
```

### **2.8.11. Error Objects**

ECMAScript v3 defines a number of classes that represent errors. The JavaScript interpreter "throws" an object of one of these types when a runtime error occurs. Each error object has a `message` property that contains an implementation-specific error message. The types of predefined error objects are `Error`, `EvalError`, `RangeError`, `ReferenceError`, `SyntaxError`, `TypeError`, and `URIError`. You can find out more about these classes in the core reference section of this book.

## **2.9 Variables**

A variable is a name associated with a value; we say that the variable stores or contains the value. Variables allow you to store and manipulate data in your programs. For example, the following line of JavaScript assigns the value 2 to a variable named `i`:

```
i = 2;
```

And the following line adds 3 to `i` and assigns the result to a new variable, `sum`:

```
var sum = i + 3;
```

### **2.9.1. Variable Typing**

An important difference between JavaScript and languages such as Java and C is that JavaScript is untyped. This means, in part, that a JavaScript variable can hold a value of any data type, unlike a Java or C variable, which can hold only the one particular type of data for which it is declared. For example, it is perfectly legal in JavaScript to assign a number to a variable and then later assign a string to that variable:

```
i = 10;  
i = "ten";
```

### **2.9.2. Variable Declaration**

Before you use a variable in a JavaScript program, you must declare it. Variables are declared with the `var` keyword, like this:



```
var i;
```

```
var sum;
```

You can also declare multiple variables with the same var keyword:

```
var i, sum;
```

And you can combine variable declaration with variable initialization:

```
var message = "hello";
```

```
var i = 0, j = 0, k = 0;
```

If you don't specify an initial value for a variable with the var statement, the variable is declared, but its initial value is undefined until your code stores a value into it.

Note that the var statement can also appear as part of the for and for/in loops, allowing you to succinctly declare the loop variable as part of the loop syntax itself. For example:

```
for(var i = 0; i < 10; i++) document.write(i, "<br>");
```

```
for(var i = 0, j=10; i < 10; i++,j--) document.write(i*j, "<br>");
```

```
for(var i in o) document.write(i, "<br>");
```

### **2.9.3. Variable Scope**

The scope of a variable is the region of your program in which it is defined. A global variable has global scope -- it is defined everywhere in your JavaScript code. On the other hand, variables declared within a function are defined only within the body of the function. They are local variables and have local scope. Function parameters also count as local variables and are defined only within the body of the function.

Within the body of a function, a local variable takes precedence over a global variable with the same name. If you declare a local variable or function parameter with the same name as a global variable, you effectively hide the global variable.

## **2.10. Statements**

A JavaScript program is simply a collection of statements, so once you are familiar with the statements of JavaScript, you can begin writing JavaScript programs.

### **2.10. Expression Statements**

The simplest kinds of statements in JavaScript are expressions that have side effects. Assignment statements are one major category of expression statements. For example:

```
s = "Hello " + name;
```

```
i *= 3;
```

```
counter++;
```

```
delete o.x;
```

### 2.10.1. Compound Statements

The comma operator can be used to combine a number of expressions into a single expression. JavaScript also has a way to combine a number of statements into a single statement, or statement block. This is done simply by enclosing any number of statements within curly braces. Thus, the following lines act as a single statement and can be used anywhere that JavaScript expects a single statement:

```
{  
    x = Math.PI;  
    cx = Math.cos(x);  
    alert("cos(" + x + ") = " + cx);  
}
```

#### **if**

The if statement is the fundamental control statement that allows JavaScript to make decisions, or, more precisely, to execute statements conditionally. This statement has two forms. The first is:

```
if (expression)  
    statement
```

For example:

```
if (username == null)    // If username is null or undefined,  
    username = "John Doe"; // define it
```

The second form of the if statement introduces an else clause that is executed when *expression* is false. Its syntax is:

```
if (expression)  
    statement1  
else  
    statement2
```

For example:

```
if (username != null)  
    alert("Hello " + username + "\nWelcome to my home page.");  
else {  
    username = prompt("Welcome!\n What is your name?");  
    alert("Hello " + username);  
}
```

Another form of if is nested if:

```
if (i == j) {  
    if (j == k) {
```

```

    document.write("i equals k");
}
}
else { // What a difference the location of a curly brace makes!
    document.write("i doesn't equal j");
}

```

#### **else if**

```

if (n == 1) {
    // Execute code block #1
}
else if (n == 2) {
    // Execute code block #2
}
else if (n == 3) {
    // Execute code block #3
}
else {
    // If all else fails, execute block #4
}

```

#### **switch**

```

switch(expression) {
    case vluel: block1; break;
    case vaue2: block2; break;
    .
    .
    default:block;
}

```

#### **Example:**

```

switch(n) {
    case 1:           // Start here if n == 1
        // Execute code block #1.
        break;        // Stop here
    case 2:           // Start here if n == 2
        // Execute code block #2.
        break;        // Stop here
}

```

```

case 3:           // Start here if n == 3
    // Execute code block #3.
    break;        // Stop here
default:         // If all else fails...
    // Execute code block #4.
    break;        // stop here
}

```

When using switch inside a function, however, you may use a return statement instead of a break statement. Both serve to terminate the switch statement and prevent execution from falling through to the next case.

Here is a more realistic example of the switch statement; it converts a value to a string in a way that depends on the type of the value:

```

function convert(x) {
    switch(typeof x) {
        case 'number':    // Convert the number to a hexadecimal integer
            return x.toString(16);
        case 'string':    // Return the string enclosed in quotes
            return '"' + x + '"';
        case 'boolean':   // Convert to TRUE or FALSE, in uppercase
            return x.toString().toUpperCase( );
        default:          // Convert any other type in the usual way
            return x.toString( )
    }
}

```

Note that in the two previous examples, the case keywords are followed by number and string literals.

This makes the JavaScript switch statement much different from the switch statement of C, C++, and Java. In those languages, the case expressions must be compile-time constants, they must evaluate to integers or other integral types, and they must all evaluate to the same type.

```

case 60*60*24:
case Math.PI:
case n+1:
case a[0]:

```

The switch statement first evaluates the expression that follows the switch keyword, and then evaluates the case expressions, in the order in which they appear, until it finds a value that matches.

The matching case is determined using the `===` identity operator, not the `==` equality operator, so the expressions must match without any type conversion.

This means that the JavaScript switch statement is not nearly as efficient as the switch statement in C, C++, and Java. Since the case expressions in those languages are compile-time constants, they never need to be evaluated at runtime as they are in JavaScript. Furthermore, since the case expressions are integral values in C, C++, and Java, the switch statement can often be implemented using a highly efficient "jump table."

Note that it is not good programming practice to use case expressions that contain side effects such as function calls or assignments, because not all of the case expressions are evaluated each time the switch statement is executed. When side effects occur only sometimes, it can be difficult to understand and predict the correct behavior of your program. The safest course is simply to limit your case expressions to constant expressions.

## **while**

General syntax:

```
while (expression)  
statement
```

Here is an example while loop:

```
var count = 0;  
while (count < 10) {  
    document.write(count + "<br>");  
    count++;  
}
```

## **do/while**

The syntax is:

```
do  
statement  
while (expression);
```

For example:

```
function printArray(a) {  
    if (a.length == 0)  
        document.write("Empty Array");  
    else {  
        var i = 0;  
        do {  
            document.write(a[i] + "<br>");
```

```

        } while (++i < a.length);
    }
}

```

There are a couple of differences between the do/while loop and the ordinary while loop. First, the do loop requires both the do keyword (to mark the beginning of the loop) and the while keyword (to mark the end and introduce the loop condition). Also, unlike the while loop, the do loop is terminated with a semicolon. This is because the do loop ends with the loop condition, rather than simply with a curly brace that marks the end of the loop body.

## **for**

The syntax of the for statement is:

```

for(initialize ; test ; increment)
    statement

```

Example 1:

```

for(var count = 0 ; count < 10 ; count++)
    document.write(count + "<br>");

```

Example 2:

```

for(i = 0, j = 10 ; i < 10 ; i++, j--)
    sum += i * j;

```

## **for/in**

The for keyword is used in two ways in JavaScript. We've just seen how it is used in the for loop. It is also used in the for/in statement. This statement is a somewhat different kind of loop with the following syntax:

```

for (variable in object)
    statement

```

Example:

```

<html>
<head><title>The Array Object</title>
<h2>An Array of Books</h2>
<script language="JavaScript">
    var book = new Array(6); // Create an Array object
    book[0] = "War and Peace"; // Assign values to its elements
    book[1] = "Huckleberry Finn";
    book[2] = "The Return of the Native";
    book[3] = "A Christmas Carol";

```

```

        book[4] = "The Yearling";
        book[5] = "Exodus";
    </script>
</head>
<body bgcolor="lightblue">
    <script language="JavaScript">
        document.write("<h3>");
        for(var i in book){
            document.write("book[" + i + "] " + book[i] + "<br>");
        }
    </script>
</body>
</html>

```

## Labels

In JavaScript any statement may be labeled by preceding it with an identifier name and a colon:

```
identifier: statement
```

The *identifier* can be any legal JavaScript identifier that is not a reserved word. Label names are distinct from variable and function names, so you do not need to worry about name collisions if you give a label the same name as a variable or function. Here is an example of a labeled while statement:

```

parser:
    while(token != null) {
        // Code omitted here
    }

```

## break

The break statement causes the innermost enclosing loop or a switch statement to exit immediately. Its syntax is simple:

```
break;
```

Because it causes a loop or switch to exit, this form of the break statement is legal only if it appears within one of these statements. Latest versions allow the break keyword to be followed by the name of a label:

```
break labelname;
```

When break is used with a label, it jumps to the end of, or terminates, the named statement, which may be any enclosing statement.

```
for(i = 0; i < a.length; i++) {
```

```

    if (a[i] == target)
        break;
}

```

You need the labeled form of the break statement only when you are using nested loops or switch statements and need to break out of a statement that is not the innermost one.

The following example shows labeled for loops and labeled break statements. See if you can figure out what its output will be:

```

outerloop:
for(var i = 0; i < 10; i++) {
    innerloop:
    for(var j = 0; j < 10; j++) {
        if (j > 3) break;           // Quit the innermost loop
        if (i == 2) break innerloop; // Do the same thing
        if (i == 4) break outerloop; // Quit the outer loop
        document.write("i = " + i + " j = " + j + "<br>");
    }
}
document.write("FINAL i = " + i + " j = " + j + "<br>");

```

## **continue**

The continue statement is similar to the break statement. Instead of exiting a loop, however, continue restarts a loop in a new iteration. The continue statement's syntax is just as simple as the break statement's:

```

continue;
continue labelname;

```

The continue statement, in both its labeled and unlabeled forms, can be used only within the body of a while, do/while, for, or for/in loop. Using it anywhere else causes a syntax error.

```

for(i = 0; i < data.length; i++) {
    if (data[i] == null)
        continue; // Can't proceed with undefined data
    total += data[i];
}

```

## **2.11. Functions**

Functions are an important and complex part of the JavaScript language. This chapter examines functions from several points of view. This chapter focuses on defining and invoking user-defined JavaScript functions. It is also important to remember that JavaScript supports quite a few built-in



functions, such as `eval( )`, `parseInt( )`, and the `sort( )` method of the `Array` class. Client-side JavaScript defines others, such as `document.write( )` and `alert( )`. Built-in functions in JavaScript can be used in exactly the same ways as user-defined functions.

### 2.11.1. Defining and Invoking Functions

This statement consists of the function keyword, followed by:

- The name of the function
- An optional comma-separated list of parameter names in parentheses
- The JavaScript statements that comprise the body of the function, contained within curly braces

#### Defining JavaScript functions

```
// A shortcut function, sometimes useful instead of document.write( )  
// This function has no return statement, so it returns no value  
function print(msg)  
{  
    document.write(msg, "<br>");  
}  
  
// A function that computes and returns the distance between two points  
function distance(x1, y1, x2, y2)  
{  
    var dx = x2 - x1;  
    var dy = y2 - y1;  
    return Math.sqrt(dx*dx + dy*dy);  
}  
  
// A recursive function (one that calls itself) that computes factorials  
// Recall that x! is the product of x and all positive integers less than it  
function factorial(x)  
{  
    if (x <= 1)  
        return 1;  
    return x * factorial(x-1);  
}
```

Once a function has been defined, it may be invoked with the `( )` operator.

```
print("Hello, " + name);  
print("Welcome to my home page!");  
total_dist = distance(0,0,2,1) + distance(2,1,3,5);
```

```
print("The probability of that is: " + factorial(39)/factorial(52));
```

Since JavaScript is an untyped language, you are not expected to specify a data type for function parameters, and JavaScript does not check whether you have passed the type of data that the function expects. If the data type of an argument is important, you can test it yourself with the `typeof` operator. JavaScript does not check whether you have passed the correct number of arguments, either. If you pass more arguments than the function expects, the extra values are simply ignored. If you pass fewer than expected, some of the parameters are given the undefined value -- which, in many circumstances, causes your function to behave incorrectly. Note that the `print( )` function does not contain a return statement, so it always returns the undefined value and cannot meaningfully be used as part of a larger expression.

### **2.11.2. Nested Functions**

Function definitions can be nested within other functions. For example:

```
function hypotenuse(a, b) {  
    function square(x) { return x*x; }  
    return Math.sqrt(square(a) + square(b));  
}
```

### **2.11.3. The Function( ) Constructor**

The function statement is not the only way to define a new function. You can define a function dynamically with the `Function( )` constructor and the `new` operator. Here is an example of creating a function in this way:

```
var f = new Function("x", "y", "return x*y;");
```

This line of code creates a new function that is more or less equivalent to a function defined with the familiar syntax: `function f(x, y) { return x*y; }`

The `Function( )` constructor expects any number of string arguments. The last argument is the body of the function -- it can contain arbitrary JavaScript statements, separated from each other by semicolons. All other arguments to the constructor are strings that specify the names of the parameters to the function being defined. If you are defining a function that takes no arguments, you simply pass a single string -- the function body -- to the constructor.

Notice that the `Function( )` constructor is not passed any argument that specifies a name for the function it creates. The unnamed functions created with the `Function( )` constructor are sometimes called anonymous functions.

### **2.11.4. Functions as Data**

The most important features of functions are that they can be defined and invoked, as shown in the previous section. Function definition and invocation are syntactic features of JavaScript and of most other programming languages. In JavaScript, however, functions are not only syntax but also data,

which means that they can be assigned to variables, stored in the properties of objects or the elements of arrays, passed as arguments to functions, and so on.

```
function square(x) { return x*x; }
```

This definition creates a new function object and assigns it to the variable `square`. The name of a function is really immaterial -- it is simply the name of a variable that holds the function. The function can be assigned to another variable and still work the same way:

```
var a = square(4); // a contains the number 16  
var b = square;   // Now b refers to the same function that square does  
var c = b(5);     // c contains the number 25
```

Functions can also be assigned to object properties rather than global variables. When we do this, we call them methods:

```
var o = new Object;  
o.square = new Function("x", "return x*x"); // Note Function( ) constructor  
y = o.square(16);                          // y equals 256
```

Functions don't even require names at all, as when we assign them to array elements:

```
var a = new Array(3);  
a[0] = function(x) { return x*x; } // Note function literal  
a[1] = 20;  
a[2] = a[0](a[1]);                // a[2] contains 400
```

The function invocation syntax in this last example looks strange, but it is still a legal use of the JavaScript `()` operator!

### **2.11.5. Function Scope: The Call Object**

The body of a JavaScript function executes in a local scope that differs from the global scope. This new scope is created by adding the call object to the front of the scope chain. Since the call object is part of the scope chain, any properties of this object are accessible as variables within the body of the function. Local variables declared with the `var` statement are created as properties of this object; the parameters of the function are also made available as properties of the object.

In addition to local variables and parameters, the call object defines one special property named `arguments`. This property refers to another special object known as the Arguments object, which is discussed in the next section. Because the `arguments` property is a property of the call object, it has exactly the same status as local variables and function parameters. For this reason, the identifier `arguments` should be considered a reserved word and should not be used as a variable or parameter name.

## 2.12. Objects

Objects are composite data types: they aggregate multiple values into a single unit and allow us to store and retrieve those values by name. Another way to explain this is to say that an object is an unordered collection of properties, each of which has a name and a value. The named values held by an object may be primitive values like numbers and strings, or they may themselves be objects.

### 2.12.1. Creating Objects

Objects are created with the `new` operator. This operator must be followed by the name of a constructor function that serves to initialize the object. For example, we can create an empty object (an object with no properties) like this:

```
var o = new Object( );
```

JavaScript supports other built-in constructor functions that initialize newly created objects in other, less trivial, ways. For example, the `Date( )` constructor initializes an object that represents a date and time:

```
var now = new Date( );           // The current date and time
var new_years_eve = new Date(2000, 11, 31); // Represents December 31, 2000
```

An object literal consists of a comma-separated list of property specifications enclosed within curly braces. Each property specification in an object literal consists of the property name followed by a colon and the property value. For example:

```
var circle = { x:0, y:0, radius:2 }
var homer = {
    name: "Homer Simpson",
    age: 34,
    married: true,
    occupation: "plant operator",
    email: "homer@simpsons.com"
};
```

### 2.12.2. Setting and Querying Properties

You normally use the `.` operator to access the value of an object's properties. The value on the left of the `.` should be a reference to an object (usually just the name of the variable that contains the object reference). The value on the right of the `.` should be the name of the property. This must be an identifier, not a string or an expression. For example:

```
// Create an object. Store a reference to it in a variable.
var book = new Object( );
// Set a property in the object.
book.title = "JavaScript: The Definitive Guide"
```

```

// Set some more properties. Note the nested objects.
book.chapter1 = new Object( );
book.chapter1.title = "Introduction to JavaScript";
book.chapter1.pages = 19;
book.chapter2 = { title: "Lexical Structure", pages: 6 };
// Read some property values from the object.
alert("Outline: " + book.title + "\n\t" +
      "Chapter 1 " + book.chapter1.title + "\n\t" +
      "Chapter 2 " + book.chapter2.title);

```

An important point to notice about this example is that you can create a new property of an object simply by assigning a value to it. Although we declare variables with the `var` keyword, there is no need (and no way) to do so with object properties. Furthermore, once you have created an object property by assigning a value to it, you can change the value of the property at any time simply by assigning a new value:

```
book.title = "JavaScript: The Rhino Book"
```

### 2.12.3. Constructors

We saw previously that you can create and initialize a new object in JavaScript by using the `new` operator in conjunction with a predefined constructor function such as `Object( )`, `Date( )`, or `Function( )`. These predefined constructors and the built-in object types they create are useful in many instances. However, in object-oriented programming, it is also common to work with custom object types defined by your program. For example, if you are writing a program that manipulates rectangles, you might want to represent rectangles with a special type, or class, of object. Each object of this `Rectangle` class would have a `width` property and a `height` property, since those are the essential defining characteristics of rectangles.

To create objects with properties such as `width` and `height` already defined, we need to write a constructor to create and initialize these properties in a new object. A constructor is a JavaScript function with two special features:

1. It is invoked through the `new` operator.
2. It is passed a reference to a newly created, empty object as the value of the `this` keyword, and it is responsible for performing appropriate initialization for that new object.

A `Rectangle` object constructor function

```

// Define the constructor.
// Note how it initializes the object referred to by "this".
function Rectangle(w, h)
{

```

```

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

```

Notice how the constructor uses its arguments to initialize properties of the object referred to by the `this` keyword. We 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.

## 2.13. Arrays

A composite data type that holds named values. An array is a data type that contains or stores numbered values. Each numbered value is called an element of the array, and the number assigned to an element is called its index. Because JavaScript is an untyped language, an element of an array may be of any type, and different elements of the same array may be of different types. Array elements may even contain other arrays, which allows you to create data structures that are arrays of arrays.

### 2.13.1. Creating Arrays

Arrays are created with the `Array( )` constructor and the `new` operator. You can invoke the `Array( )` constructor in three distinct ways.

The first way is to call it with no arguments:

```
var a = new Array( );
```

This method creates an empty array with no elements.

The second method of invoking the `Array( )` constructor allows you to explicitly specify values for the first `n` elements of an array:

```
var a = new Array(5, 4, 3, 2, 1, "testing, testing");
```

In this form, the constructor takes a list of arguments. Each argument specifies an element value and may be of any type. Elements are assigned to the array starting with element 0. The length property of the array is set to the number of arguments passed to the constructor.

The third way to invoke the `Array( )` constructor is to call it with a single numeric argument, which specifies a length:

```
var a = new Array(10);
```

Finally, array literals provide another way to create arrays. An array literal allows us to embed an array value directly into a JavaScript program in the same way that we define a string literal by placing the string text between quotation marks. To create an array literal, simply place a comma-separated list of values between square brackets. For example:

```
var primes = [2, 3, 5, 7, 11];  
var a = ['a', true, 4.78];
```

Array literals can contain object literals or other array literals:

```
var b = [[1, {x:1, y:2}], [2, {x:3, y:4}]];
```

### **2.13.2. Reading and Writing Array Elements**

You access an element of an array using the `[]` operator. A reference to the array should appear to the left of the brackets. An arbitrary expression that has a non-negative integer value should be inside the brackets. You can use this syntax to both read and write the value of an element of an array. Thus, the following are all legal JavaScript statements:

```
value = a[0];  
a[1] = 3.14;  
i = 2;  
a[i] = 3;  
a[i + 1] = "hello";
```

### **2.13.3. Adding New Elements to an Array**

In languages such as C and Java, an array has a fixed number of elements that must be specified when you create the array. This is not the case in JavaScript -- an array can have any number of elements, and you can change the number of elements at any time. To add a new element to an array, simply assign a value to it: `a[10] = 10;`

Arrays in JavaScript may be sparse. This means that array indexes need not fall into a contiguous range of numbers; a JavaScript implementation may allocate memory only for those array elements that are actually stored in the array. Thus, when you execute the following lines of code, the JavaScript interpreter will typically allocate memory only for array indexes 0 and 10,000, not for the 9,999 indexes between:

```
a[0] = 1;  
a[10000] = "this is element 10,000";
```

Note that array elements can also be added to objects:

```
var c = new Circle(1,2,3);  
c[0] = "this is an array element of an object!";
```

#### 2.13.4. Array Length

All arrays, whether created with the `Array()` constructor or defined with an array literal, have a special `length` property that specifies how many elements the array contains. More precisely, since arrays can have undefined elements, the `length` property is always one larger than the largest element number in the array. The following code illustrates:

```
var a = new Array( ); // a.length == 0 (no elements defined)
a = new Array(10);    // a.length == 10 (empty elements 0-9 defined)
a = new Array(1,2,3); // a.length == 3 (elements 0-2 defined)
a = [4, 5];           // a.length == 2 (elements 0 and 1 defined)
a[5] = -1;            // a.length == 6 (elements 0, 1, and 5 defined)
a[49] = 0;            // a.length == 50 (elements 0, 1, 5, and 49 defined)
```

Remember that array indexes must be less than  $2^{32} - 1$ , which means that the largest possible value for the `length` property is  $2^{32} - 1$ .

```
var fruits = ["mango", "banana", "cherry", "pear"];
for(var i = 0; i < fruits.length; i++)
    alert(fruits[i]);
```

This example assumes, of course, that elements of the array are contiguous and begin at element 0. If this were not the case, we would want to test that each array element was defined before using it:

```
for(var i = 0; i < fruits.length; i++)
    if (fruits[i] != undefined) alert(fruits[i]);
```

The `length` property of an array is a read/write value. If you set `length` to a value smaller than its current value, the array is truncated to the new length; any elements that no longer fit are discarded and their values are lost. If you make `length` larger than its current value, new, undefined elements are added at the end of the array to increase it to the newly specified size.

Truncating an array by setting its `length` property is the only way that you can actually shorten an array. If you use the delete operator to delete an array element, that element becomes undefined, but the `length` property does not change.

#### 2.13.5. Multidimensional Arrays

JavaScript does not support true multidimensional arrays, but it does allow you to approximate them quite nicely with arrays of arrays. To access a data element in an array of arrays, simply use the `[]` operator twice. For example, suppose the variable `matrix` is an array of arrays of numbers. Every element `matrix[x]` is an array of numbers. To access a particular number within this array, you would write `matrix[x][y]`.



### 2.13.6. Array Methods

In addition to the `[]` operator, arrays can be manipulated through various methods provided by the `Array` class. The following sections introduce these methods. Many of the methods were inspired in part by the Perl programming language; Perl programmers may find them comfortingly familiar.

#### **join( )**

The `Array.join( )` method converts all the elements of an array to strings and concatenates them. You can specify an optional string that is used to separate the elements in the resulting string. If no separator string is specified, a comma is used. For example, the following lines of code produce the string `"1,2,3"`:

```
var a = [1, 2, 3]; // Create a new array with these three elements
var s = a.join( ); // s == "1,2,3"
```

The following invocation specifies the optional separator to produce a slightly different result:

```
s = a.join(", "); // s == "1, 2, 3"
```

Notice the space after the comma. The `Array.join( )` method is the inverse of the `String.split( )` method, which creates an array by breaking up a string into pieces.

#### **reverse( )**

The `Array.reverse( )` method reverses the order of the elements of an array and returns the reversed array. It does this in place -- in other words, it doesn't create a new array with the elements rearranged, but instead rearranges them in the already existing array. For example, the following code, which uses the `reverse( )` and `join( )` methods, produces the string `"3,2,1"`:

```
var a = new Array(1,2,3); // a[0] = 1, a[1] = 2, a[2] = 3
a.reverse( );           // now a[0] = 3, a[1] = 2, a[2] = 1
var s = a.join( );      // s == "3,2,1"
```

#### **sort( )**

`Array.sort( )` sorts the elements of an array in place and returns the sorted array. When `sort( )` is called with no arguments, it sorts the array elements in alphabetical order (temporarily converting them to strings to perform the comparison, if necessary):

```
var a = new Array("banana", "cherry", "apple");
a.sort( );
var s = a.join(", "); // s == "apple, banana, cherry"
```

If an array contains undefined elements, they are sorted to the end of the array.

To sort an array into some order other than alphabetical, you must pass a comparison function as an argument to `sort( )`. This function decides which of its two arguments should appear first in the sorted array. If the first argument should appear before the second, the comparison function should return a number less than zero. If the first argument should appear after the second in the sorted

array, the function should return a number greater than zero. And if the two values are equivalent (i.e., if their order is irrelevant), the comparison function should return 0. So, for example, to sort array elements into numerical rather than alphabetical order, you might do this:

```
var a = [33, 4, 1111, 222];  
a.sort( );           // Alphabetical order: 1111, 222, 33, 4  
a.sort(function(a,b) { // Numerical order: 4, 33, 222, 1111  
    return a-b; // Returns < 0, 0, or > 0, depending on order  
});
```

Note the convenient use of a function literal in this code. Since the comparison function is used only once, there is no need to give it a name.

As another example of sorting array items, you might perform a case-insensitive alphabetical sort on an array of strings by passing a comparison function that converts both of its arguments to lowercase (with the `toLowerCase( )` method) before comparing them.

### **concat( )**

The `Array.concat( )` method creates and returns a new array that contains the elements of the original array on which `concat( )` was invoked, followed by each of the arguments to `concat( )`. If any of these arguments is itself an array, it is flattened and its elements are added to the returned array. Note, however, that `concat( )` does not recursively flatten arrays of arrays. Here are some examples:

```
var a = [1,2,3];  
a.concat(4, 5) // Returns [1,2,3,4,5]  
a.concat([4,5]); // Returns [1,2,3,4,5]  
a.concat([4,5],[6,7]) // Returns [1,2,3,4,5,6,7]  
a.concat(4, [5,[6,7]]) // Returns [1,2,3,4,5,[6,7]]
```

### **slice( )**

The `Array.slice( )` method returns a *slice*, or subarray, of the specified array. Its two arguments specify the start and end of the slice to be returned. The returned array contains the element specified by the first argument and all subsequent elements up to, but not including, the element specified by the second argument. If only one argument is specified, the returned array contains all elements from the start position to the end of the array. If either argument is negative, it specifies an array element relative to the last element in the array. An argument of -1, for example, specifies the last element in the array, and an argument of -3 specifies the third from last element of the array. Here are some examples:

```
var a = [1,2,3,4,5];  
a.slice(0,3); // Returns [1,2,3]
```

```
a.slice(3);    // Returns [4,5]  
a.slice(1,-1); // Returns [2,3,4]  
a.slice(-3,-2); // Returns [3]
```

## **2.14. The Web Browser Environment**

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

1. The Window object that serves as the global object and global execution context for client-side JavaScript code
2. The client-side object hierarchy and the document object model that forms a part of it
3. The event-driven programming model

### **2.14.1. 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 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.

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 us 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.

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. To client-side JavaScript, top-level windows and frames are essentially equivalent.

### **2.14.2. Accessing window properties and methods**

The most logical and common way to compose such references includes the window object in the reference:

```
window.propertyName  
window.methodName([parameters])
```

A window object also has a synonym when the script doing the referencing points to the window that houses the document. The synonym is self. Reference syntax then becomes

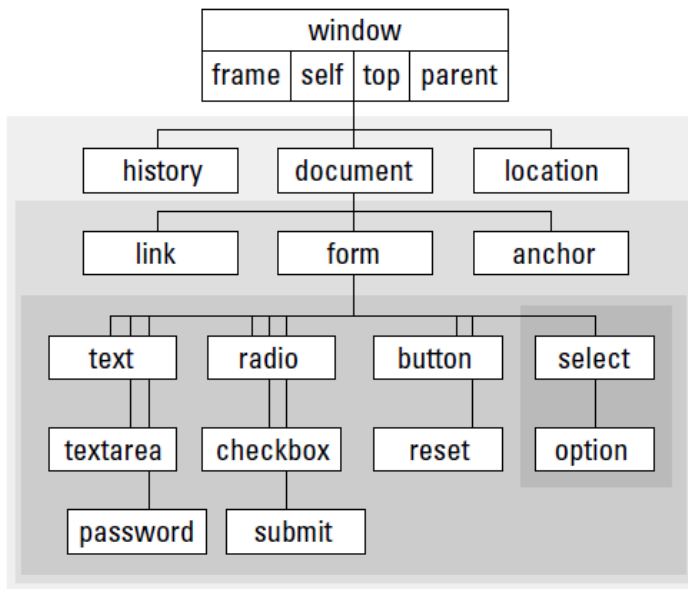
*self.propertyName*

*self.methodName([parameters])*

Since the window object is always “there” when a script runs, you could omit it from references to any objects inside that window. Therefore, the following syntax models assume properties and methods of the current window:

*propertyName*

*methodName([parameters])*



## 2.15. The Client-Side Object Hierarchy and the Document Object Model

We've seen that the Window object is the key object in client-side JavaScript. All other client-side objects are connected to this object. For example, every Window object contains 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. A Window object also contains a frames[] array that refers to the Window objects that represent the frames of the original window. Thus, document represents the Document object of the current window, and frames[1].document refers to the Document object of the second child frame of the current window.

An object referenced through the current window or through some other Window object may itself 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]*

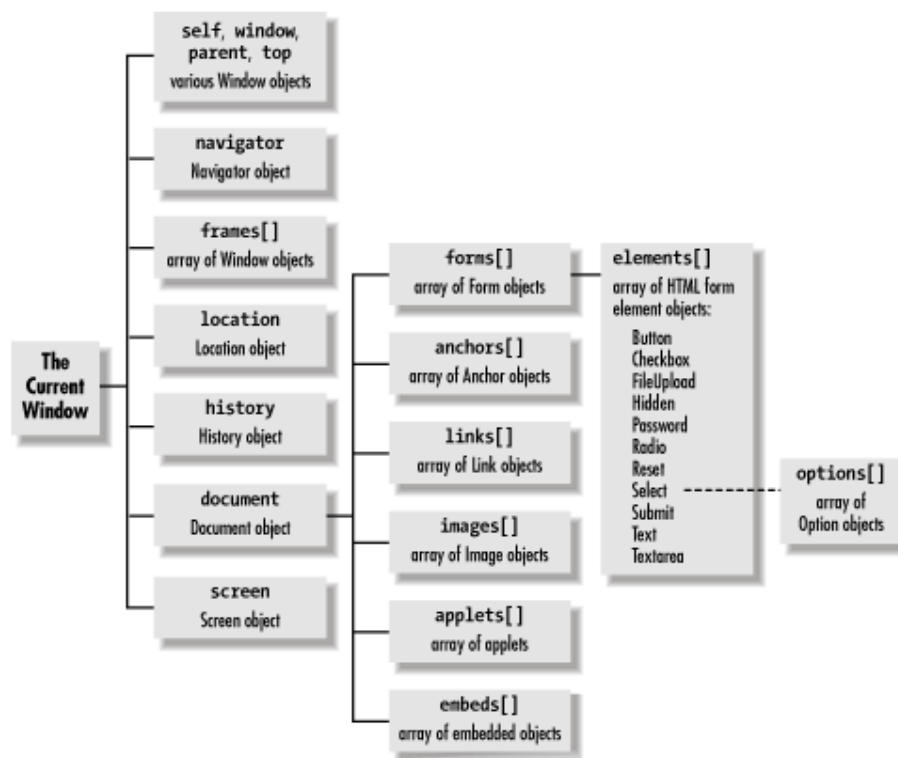
To continue with the same example, each Form object has an elements[] array containing objects that represent the various HTML form elements (input fields, buttons, etc.) that appear within the

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
```

We've seen that the Window object is the global object at the head of the scope chain and that 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.

**The client-side object hierarchy and Level 0 DOM**



## 2.16. The Event-Driven Programming Model

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. Sometimes this sort of program is useful. More often, however, we want to write dynamic programs that interact with the user. To do this, we 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, we must

define appropriate event handlers and register them with the system, so that the browser can invoke them at appropriate times.

## 2.17. Event Handlers

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 are 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="opts" value="ignore-case"
      onclick="ignore-case = this.checked;" >
```

What's of interest to us 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.

All HTML event handler attribute names begin with "on".

While you can include any number of JavaScript statements within an event handler definition, a common technique when more than one or two simple statements are required is to define the body of an event handler as a function between <script> and </script> tags. Then you can simply invoke this function from the event handler. This keeps most of your actual JavaScript code within scripts and reduces the need to mingle JavaScript and HTML.

These are the most common events:

*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.

*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.

*onmouseover* , *onmouseout*

These two event handlers are triggered when the mouse pointer moves over or out of a document element, respectively. They are used most frequently with <a> tags. If the onmouseover handler of an <a> tag returns true, it prevents the browser from displaying the URL of the link in the status line.

*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.

*onsubmit , onreset*

These event handlers are supported by the `<form>` tag and are triggered when the form is about to be submitted or reset. They can return false to cancel the submission or reset. The `onsubmit` handler is commonly used to perform client-side form validation.

## **2.18. Windows and Frames**

Most of the client-side JavaScript examples we've seen so far have involved only a single window or frame. In the real world, JavaScript applications often involve multiple windows or frames. Recall that frames within a window are represented by Window objects; JavaScript makes little distinction between windows and frames.

### **2.18.1. Relationships Between Frames**

We've already seen that the `open( )` method of the Window object returns a new Window object representing the newly created window. We've also seen that this new window has an `opener` property that refers back to the original window. In this way, the two windows can refer to each other, and each can read properties and invoke methods of the other. The same thing is possible with frames. Any frame in a window can refer to any other frame through the use of the `frames`, `parent`, and `top` properties of the Window object.

Every window has a `frames` property. This property refers to an array of Window objects, each of which represents a frame contained within the window. (If a window does not have any frames, the `frames[]` array is empty and `frames.length` is zero.) Thus, a window (or frame) can refer to its first subframe as `frames[0]`, its second subframe as `frames[1]`, and so on. Similarly, JavaScript code running in a window can refer to the third subframe of its second frame like this:

*`frames[1].frames[2]`*

Every window also has a `parent` property, which refers to the Window object in which it is contained. Thus, the first frame within a window might refer to its sibling frame (the second frame within the window) like this:

*`parent.frames[1]`*

If a window is a top-level window and not a frame, `parent` simply refers to the window itself:

*`parent == self; // For any top-level window`*

If a frame is contained within another frame that is contained within a top-level window, that frame can refer to the top-level window as `parent.parent`. Frames are typically created with `<frameset>` and `<frame>` tags.

### 2.18.2. Window and Frame Names

The second, optional argument to the `open( )` method discussed earlier is a name for the newly created window. When you create a frame with the HTML `<frame>` tag, you can specify a name with the `name` attribute. An important reason to specify names for windows and frames is that those names can be used as the value of the `target` attribute of the `<a>`, `<map>`, and `<form>` tags. This value tells the browser where you want to display the results of activating a link, clicking on an image map, or submitting a form.

For example, if you have two windows, one named `table_of_contents` and the other `mainwin`, you might have HTML like the following in the `table_of_contents` window:

```
<a href="chapter01.html" target="mainwin">  
    Chapter 1, Introduction  
</a>
```

The browser loads the specified URL when the user clicks on this hyperlink, but instead of displaying the URL in the same window as the link, it displays it in the window named `mainwin`. If there is no window with the name `mainwin`, clicking the link creates a new window with that name and loads the specified URL into it.

The `target` and `name` attributes are part of HTML and operate without the intervention of JavaScript, but there are also JavaScript-related reasons to give names to your frames. We've seen that every `Window` object has a `frames[]` array that contains references to each of its frames. This array contains all the frames in a window (or frame), whether or not they have names. If a frame is given a name, however, a reference to that frame is also stored in a new property of the parent `Window` object. The name of that new property is the same as the name of the frame. Therefore, you might create a frame with HTML like this:

```
<frame name="table_of_contents" src="toc.html">
```

Now you can refer to that frame from another, sibling frame with:

```
parent.table_of_contents
```

### 2.18.3. JavaScript in Interacting Windows

As we've seen, one frame can refer to any other frame using the `frames`, `parent`, and `top` properties. So, although JavaScript code in different frames is executed with different scope chains, the code in one frame can still refer to and use the variables and functions defined by code in another frame.

For example, suppose code in frame A defines a variable `i`:

```
var i = 3;
```

That variable is nothing more than a property of the global object -- a property of the `Window` object. Code in frame A could refer to the variable explicitly as such a property with either of these two expressions:



*window.i*

*self.i*

Now suppose that frame A has a sibling frame B that wants to set the value of the variable *i* defined by the code in frame A. If frame B just sets a variable *i*, it merely succeeds in creating a new property of its own Window object. So instead, it must explicitly refer to the property *i* in its sibling frame with code like this:

```
parent.frames[0].i = 4;
```

Recall that the function keyword that defines functions declares a variable just like the `var` keyword does. If JavaScript code in frame A declares a function *f*, that function is defined only within frame A. Code in frame A can invoke *f* like this: `f()`;

Code in frame B, however, must refer to *f* as a property of the Window object of frame A:

```
parent.frames[0].f( );
```

If the code in frame B needs to use this function frequently, it might assign the function to a variable of frame B so that it can more conveniently refer to the function: `var f = parent.frames[0].f`; Now code in frame B can invoke the function as `f()`, just as code in frame A does.

#### 2.18.4. Window Overview

We begin this chapter with an overview of some of the most commonly used properties and methods of the Window object. Later sections of the chapter explain this material in more detail. As usual, the client-side reference section contains complete coverage of Window object properties and methods. The most important properties of the Window object are the following:

***closed*** A boolean value that is true only if the window has been closed.

***defaultStatus, status*** The text that appears in the status line of the browser.

***document*** A reference to the Document object that represents the HTML document displayed in the window.

***frames[]*** An array of Window objects that represent the frames (if any) within the window.

***history*** A reference to the History object that represents the user's browsing history for the window.

***Location*** A reference to the Location object that represents the URL of the document displayed in the window.

***name*** The name of the window. Can be used with the target attribute of the HTML `<a>` tag, for example.

***opener*** A reference to the Window object that opened this one, or null if this window was opened by the user.

***parent*** If the current window is a frame, a reference to the frame of the window that contains it.

***self*** A self-referential property; a reference to the current Window object. A synonym for window.

**top** If the current window is a frame, a reference to the **Window** object of the top-level window that contains the frame. Note that **top** is different from **parent** for frames nested within other frames.

**window** A self-referential property; a reference to the current **Window** object. A synonym for **self**.

The **Window** object also supports a number of important methods:

**alert( )** , **confirm( )** , **prompt( )** Display simple dialog boxes to the user and, for **confirm( )** and **prompt( )**, get the user's response.

**close( )** Close the window.

**focus( )** , **blur( )** Request or relinquish keyboard focus for the window. The **focus( )** method also ensures that the window is visible by bringing it to the front of the stacking order.

**moveBy( )** , **moveTo( )** Move the window.

**open( )** Open a new top-level window to display a specified URL with a specified set of features.

**print( )** Print the window or frame -- same as if the user had selected the Print button from the window's toolbar **resizeBy( )** , **resizeTo( )** Resize the window.

**scrollBy( )** , **scrollTo( )** Scroll the document displayed within the window.

**setInterval( )** , **clearInterval( )** Schedule or cancel a function to be repeatedly invoked with a specified delay between invocations.

**setTimeout( )** , **clearTimeout( )** Schedule or cancel a function to be invoked once after a specified number of milliseconds.

## 2.19. Dialog Boxes

Let's start our discussion of the application of the **Window** object by covering the creation of three types of special windows known generically as dialogs. A *dialog box*, or simply *dialog*, is a small window in a graphical user interface that pops up requesting some action from a user. The three types of basic dialogs supported by JavaScript directly include alerts, confirms, and prompts.

### 2.19.1. Alert

The **Window** object's **alert()** method creates a special small window with a short string message and an OK button. The basic syntax for **alert()** is

```
window.alert(string);
```

or for shorthand,

```
alert(string);
```

as the **Window** object can be assumed.

The string passed to any dialog like an alert may be either a variable or the result of an expression. If you pass another form of data, it should be coerced into a string. All of the following examples are valid uses of the **alert()** method:

```
alert("Hi there from JavaScript! ");
```

```
alert("Hi "+username+" from Javascript");
```

```
var messageString = "Hi again!";  
alert(messageString);
```

### 2.19.2. Confirm

The **confirm()** method for the window object creates a window that displays a message for a user to respond to by clicking either an OK button to agree with the message or a Cancel button to disagree with the message. A typical rendering is shown here.

The basic syntax of the **confirm()** method is

```
window.confirm(string);
```

or simply

```
confirm(string);
```

where string is any valid string variable, literal, or expression that eventually evaluates to a string value to be used as the confirmation question.

The confirm() method returns a Boolean value that indicates whether or not the information was confirmed, true if the OK button was clicked and false if the window was closed or the Cancel button was clicked. This value can be saved to a variable, like so

```
if (confirm("Do you want ketchup on that?"))  
    alert("Pour it on!");  
else  
    alert("Hold the ketchup.");
```

### 2.19.3. Prompts

JavaScript also supports the prompt() method for the Window object. A prompt window is a small data collection dialog that prompts the user to enter a short line of data, as shown here:

The prompt() method takes two arguments. The first is a string that displays the prompt value and the second is a default value to put in the prompt window. The method returns a string value that contains the value entered by the user in the prompt. The basic syntax is shown here:

```
resultvalue = window.prompt(prompt string, default value string);
```

The shorthand **prompt()** is almost always used instead of **window.prompt()** and occasionally programmers will use only a single value in the method.

```
result = prompt("What is your favorite color?");
```

However, in most browsers you should see that a value of **undefined** is placed in the prompt line. You should set the second parameter to an empty string to keep this from happening.

```
result = prompt("What is your favorite color?", "");
```

### 2.20. The Status Line

Web browsers typically display a status line at the bottom of every window (except for those explicitly created without one), where the browser can display messages to the user. When the user

moves the mouse over a hypertext link, for example, the browser usually displays the URL to which the link points. And when the user moves the mouse over a browser control button, the browser may display a simple context help message that explains the purpose of the button. You can also make use of this status line in your own programs. Its contents are controlled by two properties of the Window object: *status* and *defaultStatus*.

```
onmouseover="self.status = 'Visit Microsoft's Home page.';return true;"
```

Although web browsers usually display the URL of a hypertext link when the user passes the mouse pointer over the link, you may have encountered some links that don't behave this way -- links that display some text other than the link's URL. This effect is achieved with the *status* property of the Window object and the *onmouseover* event handler of hypertext links:

```
<html>
<head>
<title>Generalizable window.status Property</title>
<script type="text/javascript">
function showStatus(msg) {
window.status = msg;
return true;
}
</script>
</head>
<body>
<a href="http://www.example.com" onmouseover="return showStatus('Go to my Home
page.')" onmouseout="return showStatus(')">Home</a>
<p><a href="http://mozilla.org" onmouseover="return showStatus('Visit Mozilla Home
page.')" onmouseout="return showStatus(')">Mozilla</a></p>
</body>
</html>
```

The *onmouseover* event handler in this example must return true. This tells the browser that it should not perform its own default action for the event -- that is, it should not display the URL of the link in the status line. If you forget to return true, the browser overwrites whatever message the handler displays in the status line with its own URL.

The *status* property is intended for exactly the sort of transient message we saw in the previous example. Sometimes, though, you want to display a message that is not so transient in the status line -- for example, you might display a welcome message to users visiting your web page or a simple line of help text for novice visitors. To do this, you set the *defaultStatus* property of the Window

object; this property specifies the default text displayed in the status line. That text is temporarily replaced with URLs, context help messages, or other transient text when the mouse pointer is over hyperlinks or browser control buttons, but once the mouse moves off those areas, the default text is restored.

```
<html>
<head>
<title>self Property</title>
<script type="text/javascript">
self.defaultStatus = "Welcome to my Web site.";
</script>
</head>
<body>
<a href="http://www.microsoft.com"
onmouseover="self.status = 'Visit Microsoft\'s Home page.';return true;"
onmouseout="self.status = '';return true;">Microsoft</a>
<p><a href="http://mozilla.org"
onmouseover="self.status = 'Visit Mozilla\'s Home page.';return true;"
onmouseout="self.status = self.defaultStatus;return
true;">Mozilla</a></p>
</body>
</html>
```

You might use the defaultStatus property like this to provide a friendly and helpful message to real beginners:

```
<script>
defaultStatus = "Welcome! Click on underlined blue text to navigate.";
</script>
```

## 2.21. The Navigator Object

The Window.navigator property refers to a Navigator object that contains information about the web browser as a whole, such as the version and a list of the data formats it can display. The Navigator object is named after Netscape Navigator, but it is also supported by Internet Explorer. IE also supports clientInformation as a vendor-neutral synonym for navigator. Unfortunately, Netscape and Mozilla do not support this property.

The Navigator object has five main properties that provide version information about the browser that is running:

*appName* The simple name of the web browser.

*appVersion* The version number and/or other version information for the browser.

*userAgent* The string that the browser sends in its USER-AGENT HTTP header. This property typically contains all the information in both *appName* and *appVersion*.

*appName* The code name of the browser. Netscape uses the code name "Mozilla" as the value of this property. For compatibility, IE does the same thing.

*platform* The hardware platform on which the browser is running.

```
var browser = "BROWSER INFORMATION:\n";
for(var propname in navigator) {
    browser += propname + ": " + navigator[propname] + "\n"
}
alert(browser);
```

## 2.22. Window Control Methods

The Window object defines several methods that allow high-level control of the window itself. The following sections explore how these methods allow us to open and close windows, control window position and size, request and relinquish keyboard focus, and scroll the contents of a window. We conclude with an example that demonstrates several of these features.

### 2.22.1. Opening Windows

You can open a new web browser window with the `open( )` method of the Window object. This method takes four optional arguments and returns a Window object that represents the newly opened window. The first argument to `open( )` is the URL of the document to display in the new window. The second argument to `open( )` is the name of the window. This name can be useful as the value of the `target` attribute of a `<form>` or `<a>` tag. If you specify the name of a window that already exists, `open( )` simply returns a reference to that existing window, rather than opening a new one.

The third optional argument to `open( )` is a list of features that specify the window size and GUI decorations. If you omit this argument, the new window is given a default size and has a full set of standard features: a menu bar, status line, toolbar, and so on. On the other hand, if you specify this argument, you can explicitly specify the size of the window and the set of features it includes. For example, to open a small, resizable browser window with a status bar but no menu bar, toolbar, or location bar, you could use the following line of JavaScript:

```
var w = window.open("smallwin.html", "smallwin",
    "width=400,height=350,status=yes,resizable=yes");
```

### 2.22.2. Closing Windows

Just as the `open( )` method opens a new window, the `close( )` method closes one. If we've created a Window object `w`, we can close it with: `w.close( )`;

JavaScript code running within that window itself could close it with: `window.close( )`; Again, note the explicit use of the window identifier to disambiguate the `close( )` method of the Window object from the `close( )` method of the Document object.

Most browsers allow you to automatically close only those windows that your own JavaScript code has created. If you attempt to close any other window, the user is presented with a dialog box that asks him to confirm (or cancel) that request to close the window. This precaution prevents inconsiderate scripts from writing code to close a user's main browsing window.

### **2.23. The Location Object**

The location property of a window is a reference to a Location object -- a representation of the URL of the document currently being displayed in that window. The href property of the Location object is a string that contains the complete text of the URL. Other properties of this object, such as protocol, host, pathname, and search, specify the various individual parts of the URL.

The search property of the Location object is an interesting one. It contains any portion of a URL following (and including) a question mark. This is often some sort of query string. In general, the question-mark syntax in a URL is a technique for embedding arguments in the URL. While these arguments are usually intended for CGI scripts run on a server, there is no reason why they cannot also be used in JavaScript-enabled pages.

### **2.24. The History Object**

The history property of the Window object refers to a History object for the window. The History object was originally designed to model the browsing history of a window as an array of recently visited URLs. Although its array elements are inaccessible, the History object supports three methods (which can be used by normal, unsigned scripts in all browser versions). The `back( )` and `forward( )` methods move backward or forward in a window's (or frame's) browsing history, replacing the currently displayed document with a previously viewed one. This is similar to what happens when the user clicks on the Back and Forward browser buttons. The third method, `go( )`, takes an integer argument and can skip forward or backward in the history list by multiple pages.

### **2.25. The Date Object**

Like a handful of other objects in JavaScript and the document object models, there is a distinction between the single, static Date object that exists in every window (or frame) and a date object that contains a specific date and time. The static Date object (uppercase "D") is used in only a few cases: Primarily to create a new instance of a date and to invoke a couple of methods that the Date object offers for the sake of some generic conversions.

#### **2.25.1. Creating a date object**

The statement that asks JavaScript to make an object for your script uses the special object construction keyword `new`. The basic syntax for generating a new date object is as follows:

```
var dateObjectName = new Date([parameters])
```

The date object evaluates to an object data type rather than to some string or numeric value. With the date object's reference safely tucked away in the variable name, you access all date-oriented methods in the dot-syntax fashion with which you're already familiar:

```
var result = dateObjectName.method()
```

With variables, such as result, your scripts perform calculations or displays of the date object's data (some methods extract pieces of the date and time data from the object). If you then want to put some new value into the date object (such as adding a year to the date object), you assign the new value to the object by way of the method that lets you set the value:

```
dateObjectName.method(newValue)
```

```
var oneDate = new Date() // creates object with current GMT date
```

```
var theYear = oneDate.getYear() // theYear is now storing the value 98
```

```
theYear = theYear + 1 // theYear now is 99
```

```
oneDate.setYear(theYear) // new year value now in the object
```

To create a date object for a specific date or time, you have five ways to send values as a parameter to the new Date() constructor function:

```
new Date("Month dd, yyyy hh:mm:ss")
```

```
new Date("Month dd, yyyy")
```

```
new Date(yy,mm,dd,hh,mm,ss)
```

```
new Date(yy,mm,dd)
```

```
new Date(milliseconds)
```

### **2.25.2. Date methods**

```
dateObj.getFullYear() 1970-... Specified year (NN4+, IE3/J2+)
```

```
dateObj.getYear() 70-... (See Text)
```

```
dateObj.getMonth() 0-11 Month within the year (January = 0)
```

```
dateObj.getDate() 1-31 Date within the month
```

```
dateObj.getDay() 0-6 Day of week (Sunday = 0)
```

```
dateObj.getHours() 0-23 Hour of the day in 24-hour time
```

```
dateObj.getMinutes() 0-59 Minute of the specified hour
```

```
dateObj.getSeconds() 0-59 Second within the specified minute
```

```
dateObj.getTime() 0-... Milliseconds since 1/1/70 00:00:00GMT
```

```
dateObj.getMilliseconds() 0-... Milliseconds since 1/1/70 00:00:00GMT (NN4+, IE3/J2+)
```

### **2.26. Math Object**

Whenever you need to perform math that is more demanding than simple arithmetic, look through the list of Math object methods for the solution.



## Syntax

Accessing Math object properties and methods:

*Math.property*

*Math.method(value [, value])*

The way you use the Math object in statements is the same way you use anyJavaScript object: You create a reference beginning with the Math object's name,a period, and the name of the property or method you need:

*Math.property / method([parameter]. . . [,parameter])*

Property references return the built-in values (things such as pi). Method references require one or more values to be sent as parameters of the method. Every method returns a result.

### 2.26.1. Properties

JavaScript Math object properties represent a number of valuable constant values in math.

### 2.26.2. Methods

Methods make up the balance of JavaScript Math object powers.

*Math.abs(val)* Absolute value of val

*Math.acos(val)* Arc cosine (in radians) of val

*Math.asin(val)* Arc sine (in radians) of val

*Math.atan(val)* Arc tangent (in radians) of val

*Math.atan2(val1, val2)* Angle of polar coordinates x and y

*Math.ceil(val)* Next integer greater than or equal to val

## 2.27. The Form Object

The JavaScript Form object represents an HTML form. Forms are always found as elements of the forms[] array, which is a property of the Document object. Forms appear in this array in the order in which they appear within the document. Thus, document.forms[0] refers to the first form in a document. You can refer to the last form in a document with the following: document.forms[document.forms.length-1]

The most interesting property of the Form object is the elements[] array, which contains JavaScript objects (of various types) that represent the various input elements of the form. Again, elements appear in this array in the same order they appear in the document. So you can refer to the third element of the second form in the document of the current window like this: document.forms[1].elements[2].

### 2.27.1. Naming Form Objects

*<form name="everything">*

This allowed us to refer to the Form object as: document.everything Often, you'll find this more convenient than the array notation: document.forms[0] . Furthermore, using a form name makes

your code position-independent: it works even if the document is rearranged so that forms appear in a different order.

### **2.27.2. Form Element Properties**

All (or most) form elements have the following properties in common. Some elements have other special-purpose properties that are described later, when we consider the various types of form elements individually.

**type** A read-only string that identifies the type of the form element. The third column of Table 15-1 lists the value of this property for each form element.

**Form** A read-only reference to the Form object in which this element is contained.

**name** A read-only string specified by the HTML name attribute.

**value** A read/write string that specifies the "value" contained or represented by the form element.

### **2.27.3. Form Element Event Handlers**

Most form elements support most of the following event handlers:

**onclick** Triggered when the user clicks the mouse on the element. This handler is particularly useful for Button and related form elements.

**onchange** Triggered when the user changes the value represented by the element by entering text or selecting an option, for example.

**onfocus** Triggered when the form element receives the input focus.

**onblur** Triggered when the form element loses the input focus.

## **2.28. Introduction to PHP**

PHP stands for Hypertext Preprocessor, but it is also still known around the world by its original name, Personal Home Page. It is server side programming language. It was developed by Rasmus Lerdorf in 1994.

### **2.28.1. History**

The origins of PHP date back to 1995 when an independent software development contractor named Rasmus Lerdorf developed a Perl/CGI script that enabled him to know how many visitors were reading his online resume. His script performed two tasks: logging visitor information, and displaying the count of visitors to the Web page. Because the Web as we know it today was still young at that time, tools such as these were nonexistent, and they prompted e-mails inquiring about Lerdorf's scripts. Lerdorf thus began giving away his toolset, dubbed Personal Home Page (PHP).

## **2.29. Advantages of PHP**

### **2.29.1. PHP Is Free (as in Money)**

PHP is an open source server side programming language available at free of cost that can be get easily from the market. Its coding style is quiet easy to understand and it is very efficient on multi-

platforms like Windows, Linux, and UNIX etc. It is very flexible but powerful language, most suitable for developing dynamic web pages.

### **2.29.2. PHP Is Cross-Platform**

You can use PHP with a web server computer that runs Windows, Mac OS X, Linux, Solaris, and many other versions of Unix. Plus, if you switch web server operating systems, you generally don't have to change any of your PHP programs. Just copy them from your Windows server to your Unix server, and they will still work.

While Apache is the most popular web server program used with PHP, you can also use Microsoft Internet Information Server and any other web server that supports the CGI standard. PHP also works with a large number of databases including MySQL, Oracle, Microsoft SQL Server, Sybase, and PostgreSQL. In addition, it supports the ODBC standard for database interaction.

### **2.29.3. PHP Is Widely Used**

As of March 2004, PHP is installed on more than 15 million different web sites, from countless tiny personal home pages to giants like Yahoo!. There are many books, magazines, and web sites devoted to teaching PHP and exploring what you can do with it. There are companies that provide support and training for PHP. In short, if you are a PHP user, you are not alone.

### **2.29.4. PHP Is Built for Web Programming**

Unlike most other programming languages, PHP was created from the ground up for generating web pages. This means that common web programming tasks, such as accessing form submissions and talking to a database, are often easier in PHP. PHP comes with the capability to format HTML, manipulate dates and times, and manage web cookies — tasks that are often available only as add-on libraries in other programming languages.

In terms of advantage in running, PHP does not put strain on servers. It uses its own inbuilt memory space that decreases the workload from the servers and the processing speed automatically enhances.

1. PHP has also upper hand in running multimedia files as PHP is not much dependent upon external plug-ins to run the programs.
2. And, final in terms of Budget, which is the most crucial part of the software development especially for the small business users who wants to develop money making websites to earn thick profit in minimum investments.
3. PHP (Hypertext Pre-Processor) is a server-side web programming language that is widely used for web development. However, here are many languages which are used for web development or web programming. But among all of them PHP is the most popular web scripting language. So, let us find out why PHP is widely used for web development...

4. PHP language has its roots in C and C++. PHP syntax is most similar to C and C++ language syntax. So, programmers find it easy to learn and manipulate.
5. MySQL is used with PHP as back-end tool. MySQL is the popular online database and can be interfaced very well with PHP. Therefore, PHP and MySQL are excellent choice for webmasters looking to automate their web sites.
6. PHP can run on both UNIX and Windows servers.
7. PHP also has powerful output buffering that further increases over the output flow. PHP internally rearranges the buffer so that headers come before contents.
8. PHP is dynamic. PHP works in combination of HTML to display dynamic elements on the page. PHP can be used with a large number of relational database management systems, runs on all of the most popular web servers and is available for many different operating systems.
9. PHP5 a fully object oriented language and its platform independence and speed on Linux server helps to build large and complex web applications.
10. So, in general PHP is cheap, secure, fast and reliable for developing web applications.

#### **2.29.6. What is PHP?**

- PHP stands for **PHP: Hypertext Preprocessor**
- PHP is a server-side scripting language, like ASP
- PHP scripts are executed on the server
- PHP supports many databases (MySQL, Informix, Oracle, Sybase, Solid, PostgreSQL, Generic ODBC, etc.)
- PHP is an open source software
- PHP is free to download and use

#### **2.29.7. What is a PHP File?**

- PHP files can contain text, HTML tags and scripts
- PHP files are returned to the browser as plain HTML
- PHP files have a file extension of ".php", ".php3", or ".phtml"

#### **2.29.8. Why PHP?**

- PHP runs on different platforms (Windows, Linux, Unix, etc.)
- PHP is compatible with almost all servers used today (Apache, IIS, etc.)
- PHP is FREE to download from the official PHP resource: [www.php.net](http://www.php.net)
- PHP is easy to learn and runs efficiently on the server side

#### **2.30. Basic PHP Syntax**

A PHP scripting block always starts with **<?php** and ends with **?>**. A PHP scripting block can be placed anywhere in the document. On servers with shorthand support enabled you can start a

scripting block with `<?` and end with `?>`. For maximum compatibility, we recommend that you use the standard form (`<?php`) rather than the shorthand form.

A PHP file normally contains HTML tags, just like an HTML file, and some PHP scripting code.

Below, we have an example of a simple PHP script which sends the text "Hello World" to the browser:

```
<html>
<body>
<?php
echo "Hello World";
?>
</body>
</html>
```

Each code line in PHP must end with a semicolon. The semicolon is a separator and is used to distinguish one set of instructions from another. There are two basic statements to output text with PHP: **echo** and **print**. In the example above we have used the echo statement to output the text "Hello World". **Note:** The file must have a .php extension. If the file has a .html extension, the PHP code will not be executed.

### 2.30.1. Comments in PHP

In PHP, we use `//` to make a single-line comment or `/*` and `*/` to make a large comment block.

```
<html>
<body>
<?php
//This is a comment
/*This is
a comment block */
?>
</body>
</html>
```

### 2.30.2. PHP Variables

A variable is used to store information. Variables are used for storing values, like text strings, numbers or arrays. When a variable is declared, it can be used over and over again in your script.

All variables in PHP start with a \$ sign symbol. The correct way of declaring a variable in PHP:

```
$var_name = value;
```

New PHP programmers often forget the \$ sign at the beginning of the variable. In that case it will not work.

Let's try creating a variable containing a string, and a variable containing a number:

```
<?php
$txt="Hello World!";
$x=16;
?>
```

### Storing Values in a Variable

PHP lets you store nearly anything in a variable using one of the following datatypes:

**String:** Alphanumeric characters, such as sentences or names

**Integer:** A numeric value, expressed in whole numbers

**Float:** A numeric value, expressed in real numbers (decimals)

**Boolean:** Evaluates to TRUE or FALSE (sometimes evaluates to 1 for TRUE and 0 for FALSE)

**Array:** An indexed collection of data (see the “Understanding Arrays” section later in this chapter for more information on this subject)

**Object:** A collection of data and methods

### PHP is a Loosely Typed Language

In PHP, a variable does not need to be declared before adding a value to it. In the example above, you see that you do not have to tell PHP which data type the variable is. PHP automatically converts the variable to the correct data type, depending on its value. In a strongly typed programming language, you have to declare (define) the type and name of the variable before using it. In PHP, the variable is declared automatically when you use it.

### Naming Rules for Variables

- A variable name must start with a letter or an underscore "\_"
- A variable name can only contain alpha-numeric characters and underscores (a-z, A-Z, 0-9, and \_)
- A variable name should not contain spaces. If a variable name is more than one word, it should be separated with an underscore (\$my\_string), or with capitalization (\$myString)

### 2.30.3. PHP Operators

This section lists the different operators used in PHP.

#### Arithmetic Operators

| Operator | Description    | Example    | Result |
|----------|----------------|------------|--------|
| +        | Addition       | x=2<br>x+2 | 4      |
| -        | Subtraction    | x=2<br>5-x | 3      |
| *        | Multiplication | x=4        | 20     |

|    |                              |            |     |
|----|------------------------------|------------|-----|
|    |                              | x*5        |     |
| /  | Division                     | 15/5       | 3   |
|    |                              | 5/2        | 2.5 |
| %  | Modulus (division remainder) | 5%2        | 1   |
|    |                              | 10%8       | 2   |
|    |                              | 10%2       | 0   |
| ++ | Increment                    | x=5<br>x++ | x=6 |
| -- | Decrement                    | x=5<br>x-- | x=4 |

### Assignment Operators

| Operator | Example | Is The Same As |
|----------|---------|----------------|
| =        | x=y     | x=y            |
| +=       | x+=y    | x=x+y          |
| -=       | x-=y    | x=x-y          |
| *=       | x*=y    | x=x*y          |
| /=       | x/=y    | x=x/y          |
| .=       | x.=y    | x=x.y          |
| %=       | x%=y    | x=x%y          |

### Comparison Operators

| Operator | Description                 | Example            |
|----------|-----------------------------|--------------------|
| ==       | is equal to                 | 5==8 returns false |
| !=       | is not equal                | 5!=8 returns true  |
| <>       | is not equal                | 5<>8 returns true  |
| >        | is greater than             | 5>8 returns false  |
| <        | is less than                | 5<8 returns true   |
| >=       | is greater than or equal to | 5>=8 returns false |
| <=       | is less than or equal to    | 5<=8 returns true  |

### Logical Operators

| Operator | Description | Example                                  |
|----------|-------------|--|
| &&       | and         | x=6, y=3, (x < 10 && y > 1) returns true |
|          | or          | x=6, y=3, (x==5    y==5) returns false   |
| !        | not         | x=6, y=3, !(x==y) returns true           |

#### 2.30.4. PHP If...Else Statements

Conditional statements are used to perform different actions based on different conditions. Very often when you write code, you want to perform different actions for different decisions. You can use conditional statements in your code to do this.

In PHP we have the following conditional statements:

1. **if statement** - use this statement to execute some code only if a specified condition is true
2. **if...else statement** - use this statement to execute some code if a condition is true and another code if the condition is false
3. **if...elseif...else statement** - use this statement to select one of several blocks of code to be executed
4. **switch statement** - use this statement to select one of many blocks of code to be executed

#### The if Statement

##### Syntax

if (condition) code to be executed if condition is true;

```
<html>
<body>
<?php
$d=12;
if ($d%2==0)
echo "even";
?>
</body>
</html>
```

#### The if...else Statement

##### Syntax

```
if (condition)
    code to be executed if condition is true;
else
    code to be executed if condition is false;
```

#### Example

```
<html>
<body>
<?php
```



```

$d=12;
if ($d%2==0)
    echo "even";
else
    echo "odd";
?>
</body>
</html>

```

If more than one line should be executed if a condition is true/false, the lines should be enclosed within curly braces:

```

<html>
<body>
<?php
$d=date("D");
if ($d=="Fri")
{
    echo "Hello!<br />";
    echo "Have a nice weekend!";
    echo "See you on Monday!";
}
?>
</body>
</html>

```

### **The if...elseif...else Statement**

Use the if....elseif...else statement to select one of several blocks of code to be executed.

#### **Syntax**

```

if (condition)
    code to be executed if condition is true;
elseif (condition)
    code to be executed if condition is true;
else
    code to be executed if condition is false;

```

### **2.30.5. PHP Switch Statement**

Use the switch statement to select one of many blocks of code to be executed.

## Syntax

```
switch (n)
{
  case label1:
    code to be executed if n=label1;
    break;
  case label2:
    code to be executed if n=label2;
    break;
  default:
    code to be executed if n is different from both label1 and label2;
}
```

This is how it works: First we have a single expression *n* (most often a variable), that is evaluated once. The value of the expression is then compared with the values for each case in the structure. If there is a match, the block of code associated with that case is executed. Use **break** to prevent the code from running into the next case automatically. The default statement is used if no match is found.

## Example

```
<html>
<body>
<?php
switch ($x)
{
  case 1:
    echo "Number 1";
    break;
  case 2:
    echo "Number 2";
    break;
  case 3:
    echo "Number 3";
    break;
  default:
    echo "No number between 1 and 3";
```

```
}  
?>  
</body>  
</html>
```

### 2.30.6. PHP Loops

Often when you write code, you want the same block of code to run over and over again in a row. Instead of adding several almost equal lines in a script we can use loops to perform a task like this.

In PHP, we have the following looping statements:

- **while** - loops through a block of code while a specified condition is true
- **do...while** - loops through a block of code once, and then repeats the loop as long as a specified condition is true
- **for** - loops through a block of code a specified number of times
- **foreach** - loops through a block of code for each element in an array

#### The while Loop

The while loop executes a block of code while a condition is true.

#### Syntax

```
while (condition)  
{  
    code to be executed;  
}
```

#### Example

The example below defines a loop that starts with i=1. The loop will continue to run as long as i is less than, or equal to 5. i will increase by 1 each time the loop runs:

```
<html>  
<body>  
<?php  
$i=1;  
while($i<=5)  
{  
    echo "The number is " . $i . "<br />";  
    $i++;  
}  
?>  
</body>  
</html>
```

## The do...while Statement

The do...while statement will always execute the block of code once, it will then check the condition, and repeat the loop while the condition is true.

### Syntax

```
do
{
    code to be executed;
}
while (condition);
```

### Example

The example below defines a loop that starts with i=1. It will then increment i with 1, and write some output. Then the condition is checked, and the loop will continue to run as long as i is less than, or equal to 5:

```
<html>
<body>
<?php
$i=1;
do
{
    $i++;
    echo "The number is " . $i . "<br />";
}
while ($i<=5);
?>
</body>
</html>
```

## PHP Looping - For Loops

The for loop is used when you know in advance how many times the script should run.

### Syntax

```
for (init; condition; increment)
{
    code to be executed;
}
```

Parameters:

- **init:** Mostly used to set a counter (but can be any code to be executed once at the beginning of the loop)
- **condition:** Evaluated for each loop iteration. If it evaluates to TRUE, the loop continues. If it evaluates to FALSE, the loop ends.
- **increment:** Mostly used to increment a counter (but can be any code to be executed at the end of the loop)

**Note:** Each of the parameters above can be empty, or have multiple expressions (separated by commas).

### Example

The example below defines a loop that starts with `i=1`. The loop will continue to run as long as `i` is less than, or equal to 5. `i` will increase by 1 each time the loop runs:

```
<html>
<body>
<?php
for ($i=1; $i<=5; $i++)
{
    echo "The number is " . $i . "<br />";
}
?>
</body>
</html>
```

### The foreach Loop

The foreach loop is used to loop through arrays.

#### Syntax

```
foreach ($array as $value)
{
    code to be executed;
}
```

For every loop iteration, the value of the current array element is assigned to `$value` (and the array pointer is moved by one) - so on the next loop iteration, you'll be looking at the next array value.

### Example

The following example demonstrates a loop that will print the values of the given array:

```
<html>
<body>
<?php
```

```

    $x=array("one","two","three");
    foreach ($x as $value)
    {
        echo $value . "<br />";
    }
?>
</body>
</html>

```

Output:

```

    one
    two
    three

```

## 2.31. PHP Functions

The real power of PHP comes from its functions. In PHP, there are more than 700 built-in functions. To keep the script from being executed when the page loads, you can put it into a function. A function will be executed by a call to the function. You may call a function from anywhere within a page.

### 2.31.1. Create a PHP Function

A function will be executed by a call to the function.

#### Syntax

```

function functionName()
{
    code to be executed;
}

```

PHP function guidelines:

1. Give the function a name that reflects what the function does
2. The function name can start with a letter or underscore (not a number)

#### Example

A simple function that writes my name when it is called:

```

<html>
<body>
<?php
function writeName()
{
    echo "Kai Jim Refsnes";
}

```

```

}
echo "My name is ";
writeName();
?>
</body>
</html>

```

Output: *My name is Kai Jim Refsnes*

### 2.31.2. PHP Functions - Adding parameters

To add more functionality to a function, we can add parameters. A parameter is just like a variable. Parameters are specified after the function name, inside the parentheses.

#### Example 1

The following example will write different first names, but equal last name:

```

<html>
<body>
<?php
function writeName($fname)
{
echo $fname . " Refsnes.<br />";
}
echo "My name is ";
writeName("Kai Jim");
echo "My sister's name is ";
writeName("Hege");
echo "My brother's name is ";
writeName("Stale");
?>
</body>
</html>

```

#### Example 2

The following function has two parameters:

```

<html>
<body>
<?php
function writeName($fname,$punctuation)

```

```

{
echo $fname . " Refsnes" . $punctuation . "<br />";
}
echo "My name is ";
writeName("Kai Jim", ".");
echo "My sister's name is ";
writeName("Hege", "!");
echo "My brother's name is ";
writeName("Ståle", "?");
?>
</body>
</html>

```

### 2.31.3. PHP Functions - Return values

To let a function return a value, use the return statement.

#### Example

```

<html>
<body>
<?php
function add($x,$y)
{
$total=$x+$y;
return $total;
}
echo "1 + 16 = " . add(1,16);
?>
</body>
</html>

```

Output: 1 + 16 = 17

### 2.32. PHP Strings

A string variable is used to store and manipulate text. String variables are used for values that contain characters. In this chapter we are going to look at the most common functions and operators used to manipulate strings in PHP. After we create a string we can manipulate it. A string can be used directly in a function or it can be stored in a variable. Below, the PHP script assigns the text "Hello World" to a string variable called \$txt:



```
<?php
$txt="Hello World";
echo $txt;
?>
```

The output of the code above will be: *Hello World*

### 2.32.1. The Concatenation Operator

There is only one string operator in PHP. The concatenation operator (.) is used to put two string values together. To concatenate two string variables together, use the concatenation operator:

```
<?php
$txt1="Hello World!";
$txt2="What a nice day!";
echo $txt1 . " " . $txt2;
?>
```

The output of the code above will be: *Hello World! What a nice day!*

If we look at the code above you see that we used the concatenation operator two times. This is because we had to insert a third string (a space character), to separate the two strings.

### The strlen() function

The strlen() function is used to return the length of a string.

```
<?php
echo strlen("Hello world!");
?>
```

The output of the code above will be: *12*

### The strpos() function

The strpos() function is used to search for a character/text within a string. If a match is found, this function will return the character position of the first match. If no match is found, it will return FALSE.

Let's see if we can find the string "world" in our string:

```
<?php
echo strpos("Hello world!","world");
?>
```

The output of the code above will be: *6*

## 2.33 PHP Arrays

An array stores multiple values in one single variable.

### 2.33.1. What is an Array?

A variable is a storage area holding a number or text. The problem is, a variable will hold only one value. An array is a special variable, which can store multiple values in one single variable.

An array can hold all your variable values under a single name. And you can access the values by referring to the array name. Each element in the array has its own index so that it can be easily accessed. In PHP, there are three kind of arrays:

- **Numeric array** - An array with a numeric index
- **Associative array** - An array where each ID key is associated with a value
- **Multidimensional array** - An array containing one or more arrays

### 2.33.2. Numeric Arrays

A numeric array stores each array element with a numeric index. There are two methods to create a numeric array.

1. In the following example the index are automatically assigned (the index starts at 0):

```
$cars=array("Suzuki","Volvo","BMW","Toyota");
```

2. In the following example we assign the index manually:

```
$cars[0]="Suzuki";
```

```
$cars[1]="Volvo";
```

```
$cars[2]="BMW";
```

```
$cars[3]="Toyota";
```

#### Example

In the following example you access the variable values by referring to the array name and index:

```
<?php
$cars[0]="Suzuki";
$cars[1]="Volvo";
$cars[2]="BMW";
$cars[3]="Toyota";
echo $cars[0] . " and " . $cars[1] . " are cars.";
?>
```

The code above will output: *Suzuki and Volvo are cars.*

#### Example 2

This example is the same as example 1, but shows a different way of creating the array:

```
$ages['Peter'] = "32";
```

```
$ages['Quagmire'] = "30";
```

```
$ages['Joe'] = "34";
```

The ID keys can be used in a script:

```
<?php
```

```
$ages['Peter'] = "32";  
$ages['Quagmire'] = "30";  
$ages['Joe'] = "34";  
echo "Peter is " . $ages['Peter'] . " years old."  
?>
```

*The code above will output: Peter is 32 years old.*

## **2.34. Understanding OOP Concepts**

This section introduces the primary concepts of object-oriented programming and explores how they interact.

- Classes, which are the "blueprints" for an object and are the actual code that defines the properties and methods.
- Objects, which are running instances of a class and contain all the internal data and state information needed for your application to function.
- Inheritance, which is the ability to define a class of one kind as being a sub-type of a different kind of class (much the same way a square is a kind of rectangle).
- Interfaces, which are contracts between unrelated objects to perform a common function.
- Encapsulation, which is the capability of an object to protect access to its internal data.

Along the way, we'll discuss polymorphism, which allows a class to be defined as being a member of more than one category of classes (just like a car is "a thing with an engine" and "a thing with wheels").

### **2.34.1 Classes**

In the real world, objects have characteristics and behaviors. A car has a color, a weight, a manufacturer, and a gas tank of a certain volume. Those are its characteristics. A class is a unit of code, composed of variables and functions, which describes the characteristics and behaviors of all the members of a set. A class called Car, for example, would describe the properties and methods common to all cars. In OOP terminology, the characteristics of a class are known as its properties. Properties have a name and a value.

### **2.34.2. Objects**

To begin with, you can think of a class as a blueprint for constructing an object. In much the same way that many houses can be built from the same blueprint, you can build multiple instances of an object from its class. But the blueprint doesn't specify things like the color of the walls, or type of flooring. It merely specifies that those things will exist.

Instantiating an object requires two things:

- A memory location into which to load the object. This is automatically handled for you by PHP.

- The data that will populate the values of the properties. This data could come from a database, a flat text file, another object, or some other source.

A class never has property values or state. Only objects can. You have to use the blueprint to build the house before you can give it wallpaper or vinyl siding. Similarly, you have to instantiate an object from the class before you can interact with its properties or invoke its methods.

### 2.34.3. Creating a Class

Let's start with a simple example. Save the following in a file called class.Demo.php:

```
<?php
Class Demo {
}
?>
```

And there you have it—the Demo class. Not terribly exciting just yet, but this is the basic syntax for declaring a new class in PHP. Use the keyword `class` to let PHP know you're about to define a new class. Follow that with the name of the class and braces to indicate the start and end of the code for that class.

You can instantiate an object of type Demo like this:

```
<?php
$objDemo=new Demo();
?>
```

### 2.34.4. Adding a Method

Here's an example:

```
<?php
class Demo {
    function sayHello($name) {
        print "Hello $name!";  }
}
?>
```

An object derived from your class is now capable of printing out a greeting to anyone who invokes the `sayHello` method. To invoke the method on your `$objDemo` object, you need to use the operator `->` to access the newly created function. Save the following code in a file called `testdemo.php`:

```
<?php
$objDemo = new Demo();
$objDemo->sayHello('Steve');
?>
```

The object is now capable of printing a friendly greeting! The -> operator is used to access all methods and properties of your objects. For those who have had exposure to OOP in other programming languages, please note that the -> operator is always used to access the methods and properties of an object. PHP does not use the dot operator (.) in its OO syntax at all.

### 2.34.5. Adding a Property

Adding a property to your class is as easy as adding a method. You just declare a variable inside the class to hold the value of the property. In procedural code, if you want to store a value, you assign that value to a variable. In OOP, if you want to store the value of a property, you also use a variable. This variable is declared at the top of the class declaration, inside the braces that bracket the class's code. The name of the variable is the name of the property. If the variable is called \$color, you have a property called color.

Open the class.Demo.php file and add the highlighted code:

```
<?php
class Demo {
    public $name;
    function sayHello() {
        print "Hello $this->name!";
    }
}
?>
```

Adding this new variable, called \$name, is all you have to do to create a name property of the Demo class. To use this property, the same -> operator that you saw earlier is used, along with the name of the property. The rewritten sayHello method shows how to access the value of this property.

```
<?php
$objDemo = new Demo();
$objDemo->name = 'Steve';
$objAnotherDemo = new Demo();
$objAnotherDemo->name = 'Ed';
$objDemo->sayHello();
$objAnotherDemo->sayHello();
?>
```

## 2.35. PHP Forms and User Input

The PHP \$\_GET and \$\_POST variables are used to retrieve information from forms, like user input.

### 2.35.1. PHP Form Handling

The most important thing to notice when dealing with HTML forms and PHP is that any form element in an HTML page will **automatically** be available to your PHP scripts.

### Example

The example below contains an HTML form with two input fields and a submit button:

```
<html>
<body>
<form action="welcome.php" method="post">
Name: <input type="text" name="fname" />
Age: <input type="text" name="age" />
<input type="submit" />
</form>
</body>
</html>
```

When a user fills out the form above and click on the submit button, the form data is sent to a PHP file, called "welcome.php":

"welcome.php" looks like this:

```
<html>
<body>
Welcome <?php echo $_POST["fname"]; ?>!<br />
You are <?php echo $_POST["age"]; ?> years old.
</body>
</html>
```

Output could be something like this: *Welcome John! You are 28 years old.*

### 2.35.2. Form Validation

User input should be validated on the browser whenever possible (by client scripts). Browser validation is faster and reduces the server load. You should consider server validation if the user input will be inserted into a database. A good way to validate a form on the server is to post the form to itself, instead of jumping to a different page. The user will then get the error messages on the same page as the form. This makes it easier to discover the error.

### PHP \$\_GET Variable

The predefined \$\_GET variable is used to collect values in a form with method="get". Information sent from a form with the GET method is visible to everyone (it will be displayed in the browser's address bar) and has limits on the amount of information to send.

### Example

```
<form action="http://localhost/get.php" method="get">
```

```
Name: <input type="text" name="fname" />
Age: <input type="text" name="age" />
<input type="submit" />
</form>
```

When the user clicks the "Submit" button, the URL sent to the server could look something like this: *http://www.w3schools.com/get.php?fname=Peter&age=37*

The "get.php" file can now use the \$\_GET variable to collect form data (the names of the form fields will automatically be the keys in the \$\_GET array):

```
Welcome
<?php
echo $_GET["fname"]; ?>.<br />
You are <?php echo $_GET["age"];
?> years old!
```

### **When to use method="get"?**

When using method="get" in HTML forms, all variable names and values are displayed in the URL.

**Note:** This method should not be used when sending passwords or other sensitive information!

The get method is not suitable for very large variable values. It should not be used with values exceeding 2000 characters.

### **PHP \$\_POST Variable**

The predefined \$\_POST variable is used to collect values from a form sent with method="post".

Information sent from a form with the POST method is invisible to others and has no limits on the amount of information to send.

**Note:** However, there is an 8 Mb max size for the POST method, by default (can be changed by setting the post\_max\_size in the php.ini file).

### **Example**

```
<form action="http://localhost/post.php" method="post">
Name: <input type="text" name="fname" />
Age: <input type="text" name="age" />
<input type="submit" />
</form>
```

When the user clicks the "Submit" button, the URL will look like this:

*http://www.w3schools.com/post.php*

The "post.php" file can now use the \$\_POST variable to collect form data (the names of the form fields will automatically be the keys in the \$\_POST array):

```

Welcome <?php
    echo $_POST["fname"];
?>!
<br />
You are
<?php
    echo $_POST["age"];
?> years old.

```

### **When to use method="post"?**

Information sent from a form with the POST method is invisible to others and has no limits on the amount of information to send. However, because the variables are not displayed in the URL, it is not possible to bookmark the page.

### **The PHP \$\_REQUEST Variable**

The predefined \$\_REQUEST variable contains the contents of both \$\_GET, \$\_POST, and \$\_COOKIE. The \$\_REQUEST variable can be used to collect form data sent with both the GET and POST methods.

### **Example**

```

Welcome <?php echo $_REQUEST["fname"]; ?>!
<br />
You are <?php echo $_REQUEST["age"]; ?> years old.

```

### **Echo and Print**

```

<?php
    print("Some text.");
?>

```

This code produces the following output if you reload test.php. *Some text.* The echo() Statement The most common method of generating output is probably the echo() statement. It differs slightly from print() in that it can accept multiple arguments. Consider this prototype:

```
void echo ( string $arg1 [, string $... ] )
```

The echo() statement accepts one or more arguments, separated by commas, and outputs all of the arguments to the browser in succession. Unlike print(), echo() does not return a value—the void keyword in the prototype tells it not to. Because echo() is also a language construct, the parentheses are optional and generally omitted. Add the following code to test.php:

```
<?php echo "Hello ", "world!"; ?>
```

The preceding snippet produces this output: *Hello world!*

Your two strings are added together as arguments to the echo() statement, producing one string



that ends up being passed to the browser. The same approach works for variables:

```
<?php
$foo = "Hello ";
$bar = "world!";
echo $foo, $bar;
?>
```

This produces the same output as above: *Hello world!*

### The printf() Statement

The next statement, printf(), gives you more fine-grained control over your output, allowing you to define the format of data that will be sent to the browser. You can think of this statement as meaning “print formatted.”

```
<?php
printf("PHP is %s!", "awesome");
?>

<?php
$amt1 = 2.55;
$amt2 = 3.55;
$total = $amt1 + $amt2;
echo 'The total cost is $', $total;
?>
```

You might expect to see this sentence when you run your code: *The total cost is \$6.10*.

However, what you see when you run the code is this: *The total cost is \$6.1*

For obvious reasons, this isn’t what you want to happen if you’re trying to display a price. Fortunately, this is a case where printf() comes in handy; simply add the following code to test.php:

```
<?php
$amt1 = 2.55;
$amt2 = 3.55;
$total = $amt1 + $amt2;
printf('The total cost is $%.2f', $total);
?>
```

Saving and reloading produces the desired result: *The total cost is \$6.10*

## 2.36. Databases

As you might expect, the theory and application of database development is a wide field. However, to start using a database, you need only understand a few basic concepts and operations. The most

common type of database in use today is relational databases. In the most basic view, a relational database is simply a table of rows and columns.

The basic database operations include creating tables, defining columns, adding rows, deleting rows, changing values of a row, and searching for rows that match some condition.

Almost always, a database is accessed through a database server: a machine running the database software. Hence, to use a database, a connection must be established to such a server. After a connection is made, the database is sent commands, called queries, to which the database returns a result, usually in tabular form. Different database systems implement different methods of communicating with them. Today, all readily accessible relational database systems use a language known as SQL. Each system has its own variation on the basic SQL syntax, however, and these need to be known to you to operate within that database.

For PHP to access a database, an interface library must be provided. Fortunately, interfaces for most of the popular database systems exist, including Oracle, Sybase, MySQL, Microsoft SQL Server, and PostgreSQL.

Most Web applications:

1. Retrieve information from a database to alter their on-screen display
2. Store user data such as orders, tracking, address, credit card, etc. in a database

### **2.36.1 PHP: Built-in Database Access**

- PHP provides built-in database connectivity for a wide range of databases:
  - MySQL, PostgreSQL, Oracle, Berkeley DB, Informix, mSQL, Lotus Notes, and more
  - Starting support for a specific database may involve PHP configuration steps
- Another advantage of using a programming language that has been designed for the creation of web apps.
- Support for each database is described in the PHP manual.

### **2.36.2. High-Level Process of Using MySQL from PHP**

- Create a database connection
- Select database you wish to use
- Perform a SQL query
- Do some processing on query results
- Close database connection

### **2.36.3. Creating Database Connection**

- Use either `mysql_connect` or `mysql_pconnect` to create database connection
  - `mysql_connect`: connection is closed at end of script (end of page)

- `mysql_pconnect`: creates persistent connection connection remains even after end of the page

➤ Parameters

- Server – hostname of server
- Username – username on the database
- Password – password on the database
- New Link (`mysql_connect` only) – reuse database connection created by previous call to `mysql_connect`
- Client Flags
  - `MYSQL_CLIENT_SSL` :: Use SSL
  - `MYSQL_CLIENT_COMPRESS` :: Compress data sent to MySQL

#### 2.36.4. Selecting a Database

➤ `mysql_select_db()`

- Pass it the database name

➤ Related:

- `mysql_list_dbs()`
  - List databases available
- `Mysql_list_tables()`
  - List database tables available

#### 2.36.5. Perform SQL Query

➤ Create query string

- `$query = 'SQL formatted string'`
- `$query = 'SELECT * FROM table'`

➤ Submit query to database for processing

- `$result = mysql_query($query);`
- For UPDATE, DELETE, DROP, etc, returns TRUE or FALSE
- For SELECT, SHOW, DESCRIBE or EXPLAIN, `$result` is an identifier for the results, and does not contain the results themselves
  - `$result` is called a “resource” in this case
  - A result of FALSE indicates an error

➤ If there is an error

- `mysql_error()` returns error string from last MySQL call

#### 2.36.6. Process Results

Many functions exist to work with database results

➤ `mysql_num_rows()`

- Number of rows in the result set
  - Useful for iterating over result set
- `mysql_fetch_array()`
- Returns a result row as an array
  - Can be associative or numeric or both (default)
  - `$row = mysql_fetch_array($result);`
  - `$row['column name'] ::` value comes from database row with specified column name
  - `$row[0] ::` value comes from first field in result set

Process Results Loop - Easy loop for processing results:

```
$result = mysql_query($qstring);
$num_rows = mysql_num_rows($result);
for ($i=0; $i<$num_rows; $i++) {
    $row = mysql_fetch_array($result);
    // take action on database results here
}
```

### 2.36.7. Closing Database Connection

- `mysql_close()`
- Closes database connection
  - Only works for connections opened with `mysql_connect()`
  - Connections opened with `mysql_pconnect()` ignore this call
  - Often not necessary to call this, as connections created by `mysql_connect` are closed at the end of the script anyway

### 2.37 Client-server model

A network architecture in which each computer or process on the network is either a client or a server. Servers are powerful computers or processes dedicated to managing disk drives (file servers), printers (print servers), or network traffic (network servers ). Clients are PCs or workstations on which users run applications. Clients rely on servers for resources, such as files, devices, and even processing power.

Another type of network architecture is known as a *peer-to-peer* architecture because each node has equivalent responsibilities. Both client/server and peer-to-peer architectures are widely used, and each has unique advantages and disadvantages.

Client-server architectures are sometimes called two-tier *architectures*.

### 2.38 Introduction to CGI

"CGI" stands for "Common Gateway Interface." CGI is one method by which a web server can obtain data from (or send data to) databases, documents, and other programs, and present that data

to viewers via the web. More simply, a CGI is a program intended to be run on the web. A CGI program can be written in any programming language, but Perl is one of the most popular, and for this book, Perl is the language we'll be using.

CGI helps to create interactive web pages. To create an interactive web page, html elements are used to display a form that accepts a client's input and passes this to special computer programs on the web server. These computer programs process a client's input and return requested information, usually in the form of a webpage constructed by the computer program. These programs are called as gateways because they act as a medium between the web server and an external source of information, such as a database. Gateway programs exchange information with the web server using a standard known as The CGI.

Some common uses of CGI include:

1. Gathering user feedback about a product line through an html form
2. Querying a database and rendering the result as an html document.

The web server and the CGI program normally run on the same computer, on which the web server resides. Depending on the type of request from the browser, the server either provides a document from a directory in the file system or executes a CGI program.

The purpose of the CGI program is the creation of dynamic HTML on demand from a client browser. The sequence events for creating a dynamic html document are as follows:

- A client makes an http request by means of a URL. This URL could be typed into the location window of a browser, be a hyperlink or be specified in the action attribute of an HTML form tag.
- From the URL the web server determines that it should activate the CGI script referenced in the URL and send any parameters passed via the URL to that script.
- The CGI script processes the parameters passed, then based on these parameters, returns HTML to the web server. The web server in turn adds a MIME header and returns the HTML text to the web browser.
- The web browser then renders and displays the HTML document received from the web server.

### **2.38.1. How information is transferred from the web browser to the CGI program**

Web browser uses the method attribute of the form tag to determine how to send the HTML form's data to the web server. There are two values that can be passed to the method attribute, they are:

#### **The GET method**

The GET method uses the HTTP GET command to submit the data to the web server. For using this browser must encode all the forms data into the URL.

The key features of GET method are:

- The values of all the fields are concatenated and passed to the URL specified in the action attribute of the form tag. Each field's value appears in the name-value format.
- Any character with a special meaning in the form's data is encoded using a special encoding scheme commonly referred to as URL encoding.
- In this scheme a space is replaced by a plus sign (+), fields are separated by (&), and any non-alphanumeric character is replaced by a %xx code (xx – hexadecimal representation of character).

Since data is submitted to the web server using an encoded URL the amount data that can be streamed to the web server is limited to only 2 Kbytes.

### **The POST method**

In the post method the POST command is used, where form's data is included to the body of that command. The POST method can handle any amount of data, because the browser sends the data as a separate text data stream to the web server. This method is used to send large amount of data to the server.

#### **2.38.2. How a CGI URL is interpreted by the web server**

The web server must be configured to recognize an HTTP request for a CGI program. Configuration involves information about the directory where CGI programs reside. The web server expects the CGI program's name to appear immediately following CGI directory (/cgi-bin/).

### **2.39. Environment Variables**

In the case of CGI, environment variables are known to the server. CGI is used to pass data about an HTTP request from the web server to a specific CGI program. These variables are accessible to both the web server and any CGI program invoked. These variables may also be set or assigned their values when the web server actually executes a CGI program.

To communicate with the CGI program, the web server sets up a number of environment variables with useful information. Eg. The REQUEST\_METHOD environment variable indicates the data submission method, whether it is GET or POST.

Thus environment variables provide a convenient mechanism to transfer information to a CGI program received from a browser. The requirement of a specific web server controls how CGI programs access variables. If a variable does not have a value, this indicates a zero length value (NULL). If the variable is not defined also it contains NULL.

#### **2.39.1. List of CGI Environment Variables**

GATEWAY\_INTERFACE - The revision of the Common Gateway Interface that the server uses.

SERVER\_NAME - The server's hostname or IP address.

SERVER\_SOFTWARE - The name and version of the server software that is answering the client request.

SERVER\_PROTOCOL - The name and revision of the information protocol the request came in with.

SERVER\_PORT - The port number of the host on which the server is running.

REQUEST\_METHOD - The method with which the information request was issued.

PATH\_INFO - Extra path information passed to a CGI program.

PATH\_TRANSLATED - The translated version of the path given by the variable PATH\_INFO.

SCRIPT\_NAME - The virtual path (e.g., */cgi-bin/program.pl*) of the script being executed.

DOCUMENT\_ROOT - The directory from which Web documents are served.

QUERY\_STRING - The query information passed to the program. It is appended to the URL with a "?".

REMOTE\_HOST - The remote hostname of the user making the request.

REMOTE\_ADDR - The remote IP address of the user making the request.

AUTH\_TYPE - The authentication method used to validate a user.

REMOTE\_USER - The authenticated name of the user.

REMOTE\_IDENT - The user making the request. This variable will only be set if NCSA *IdentityCheck* flag is enabled, and the client machine supports the RFC 931 identification scheme (ident daemon).

CONTENT\_TYPE - The MIME type of the query data, such as "text/html".

CONTENT\_LENGTH - The length of the data (in bytes or the number of characters) passed to the CGI program through standard input.

HTTP\_FROM - The email address of the user making the request. Most browsers do not support this variable.

HTTP\_ACCEPT - A list of the MIME types that the client can accept.

HTTP\_USER\_AGENT - The browser the client is using to issue the request.

HTTP\_REFERER - The URL of the document that the client points to before accessing the CGI program.

## **2.40. How a CGI program returns information to the server**

CGI program always returns information to the web server by writing to standard output. In other words if a CGI program wants to return an HTML document, the program must write that document to standard output. The web server then processes that output and sends the data back to the browser that had submitted the request. The CGI program adds appropriate header information to its output and sends this to the web server so that the web server knows what kind of data it is streaming back to a browser.

### **2.40.1. Processing HTML form information in a CGI program**

A CGI program needs to be able to access data returned by the browser, then process it some way before generating any meaningful output. When a browser submits data via the GET method, the CGI program obtains its information through the QUERY\_STRING environment variable.

If the data is submitted via POST method, the CGI program obtains information through standard input.

**The basic steps followed by a CGI program irrespective of the methods are:**

1. Check the REQUEST\_METHOD environment variable to determine whether the request is GET or POST.
2. If the method is GET, use the value of the QUERY\_STRING environment variable as the input. Also check for any path information in the PATH\_INFO environment variable.
3. If the method is POST, get the length of the input (in bytes) from the CONTENT\_LENGTH environment variable. Then read that many bytes from standard input.
4. Extract the name-value pairs for various fields by splitting the input data at the (&) character, which separates the name from the value.
5. In each name value pair convert all + to spaces
6. In each name value pair convert all %xx sequences to ASCII characters
7. Save the name-value pairs of specific fields for use later.

## **2.41 Programming in CGI**

A CGI program is a computer program that is started and run by a web server in response to an HTTP request. It processes data submitted to the server by a browser. The HTML form's action attribute specifies the name of the CGI program (including the TCP/IP address server where the program resides).

Example

```
<form method=GET action=http://www.abc.com/cgi-bin/pg1.cgi>
```

CGI programs can be developed using C, C++, VB Script, PERL, Python, etc.

### **Using a C program as a CGI script**

In order to set up a C program as a CGI script, it needs to be turned into a binary executable program. This is often problematic, since people largely work on Windows whereas servers often run some version of UNIX or Linux. The system where you develop your program and the server where it should be installed as a CGI script may have quite different architectures, so that the same executable does not run on both of them.

This may create an unsolvable problem. If you are not allowed to log on the server and you cannot use a binary-compatible system (or a cross-compiler) either, you are out of luck. Many servers, however, allow you log on and use the server in interactive mode, as a "shell user," and contain a C compiler.



You need to **compile and load** your C program on the **server** (or, in principle, on a system with the same architecture, so that binaries produced for it are executable on the server too). Normally, you would proceed as follows:

- Compile and test the C program in normal interactive use.
- Make any changes that might be needed for use as a CGI script. The program should read its input according to the intended form submission method. Using the default GET method, the input is to be read from the environment variable. `QUERY_STRING`. (The program may also read data from files—but these must then reside on the server.) It should generate output on the standard output stream (stdout) so that it starts with suitable HTTP headers. Often, the output is in HTML format.
- Compile and test again. In this testing phase, you might set the environment variable `QUERY_STRING` so that it contains the test data as it will be sent as form data. E.g., if you intend to use a form where a field named `foo` contains the input data, you can give the command `setenv QUERY_STRING "foo=42"` (when using the `tcsh` shell) or `QUERY_STRING="foo=42"` (when using the `bash` shell).
- Check that the compiled version is in a format that works on the server. This may require a recompilation. You may need to log on into the server computer (using Telnet, SSH, or some other terminal emulator) so that you can use a compiler there.
- Upload the compiled and loaded program, i.e. the executable binary program (and any data files needed) on the server.
- Set up a simple HTML document that contains a form for testing the script, etc.

You need to put the executable into a suitable directory and name it according to server-specific conventions. Even the compilation commands needed here might differ from what you are used to on your workstation. For example, if the server runs some flavor of Unix and has the Gnu C compiler available, you would typically use a compilation command like `gcc -o mult.cgi mult.c` and then move (`mv`) `mult.cgi` to a directory with a name like `cgi-bin`. Instead of `gcc`, you might need to use `cc`. You really need to check local instructions for such issues.

The **filename extension** `.cgi` has no fixed meaning in general. However, there can be *server-dependent* (and operating system dependent) rules for naming executable files. *Typical* extensions for executables are `.cgi` and `.exe`.

### **The Hello world test**

As usual when starting work with some new programming technology, you should probably first make a trivial program work. This avoids fighting with many potential problems at a time and concentrating first on the issues specific to the environment, here CGI.

You could use the following program that just prints Hello world but preceded by HTTP headers as required by the CGI interface. Here the header specifies that the data is plain ASCII text.

```
#include <stdio.h>

int main(void) {
    printf("Content-Type: text/plain;charset=us-ascii\n\n");
    printf("Hello world\n\n");
    return 0;
}
```

After compiling, loading, and uploading, you should be able to test the script simply by entering the URL in the browser's address bar. You could also make it the destination of a normal link in an HTML document. The URL of course depends on how you set things up; the URL for my installed Hello world script is the following: <http://localhost/cgi-bin/hellow.exe>

## 2.42. Encoding and decoding form data

For forms that use METHOD="GET" (as our simple example above uses, since this is the default), CGI specifications say that the data is passed to the script or program in an environment variable called QUERY\_STRING (the part of the URL after ?).

It depends on the scripting or programming language used how a program can access the value of an environment variable. In the C language, you would use the library function getenv (defined in the standard library stdlib) to access the value as a string. You might then use various techniques to pick up data from the string, convert parts of it to numeric values, etc.

The *output* from the script or program to “primary output stream” (such as stdout in the C language) is handled in a special way. Effectively, it is directed so that it gets sent back to the browser. Thus, by writing a C program that it writes an HTML document onto its standard output, you will make that document appear on user's screen as a response to the form submission.

In this case, the source program in C is the following:

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    char *data;
    long m,n;
    printf("Content-Type:text/html\n\n");
    printf("<TITLE>Multiplication results</TITLE>\n");
    printf("<H3>Multiplication results</H3>\n");
    data = getenv("QUERY_STRING");
```

```

if(data == NULL)
    printf("<P>Error! Error in passing data from form to script.");
else if(sscanf(data,"m=%ld&n=%ld",&m,&n)!=2)
    printf("<P>Error! Invalid data. Data must be numeric.");
else
    printf("<P>The product of %ld and %ld is %ld.",m,n,m*n);
return 0;
}

```

Note: The first printf function call prints out data that will be sent by the server as an HTTP header. This is required for several reasons, including the fact that a CGI script can send any data (such as an image or a plain text file) to the browser, not just HTML documents.

I have compiled this program and saved the executable program under the name *formget.exe* in my directory for CGI scripts at *localhost*. This implies that *any* form with action="http://localhost/cgi-bin/formget.exe" will, when submitted, be processed by that program.

Consequently, anyone could write a form of his own with the same ACTION attribute and pass whatever data he likes to my program. Therefore, the program needs to be able to **handle any data**. Generally, you need to check the data before starting to process it.

The form used for datasubmission:

```

<form action="http://localhost/cgi-bin/formget.exe">
<div><label>Multiplicand 1: <input name="m" size="5"></label></div>
<div><label>Multiplicand 2: <input name="n" size="5"></label></div>
<div><input type="submit" value="Multiply!"></div>
</form>

```

## Using METHOD="POST"

### The idea of METHOD="POST"

Let us consider next a different processing for form data. Assume that we wish to write a form that takes a line of text as input so that the form data is sent to a CGI script that *appends the data to a text file* on the server.

For forms that use METHOD="POST", CGI specifications say that the data is passed to the script or program in the standard input stream (stdin), and the length (in bytes, i.e. characters) of the data is passed in an environment variable called CONTENT\_LENGTH.

### Reading input

Reading from standard input sounds probably simpler than reading from an environment variable, but there are complications. The server is *not* required to pass the data so that when the CGI script tries to read more data than there is, it would get an end of file indication! That is, if you read e.g.

using the `getchar` function in a C program, it is *undefined* what happens after reading all the data characters; it is not guaranteed that the function will return EOF.

When reading the input, the program must not try to read more than `CONTENT_LENGTH` characters.

### **Sample program: accept and append data**

A relatively simple C program for accepting input via CGI and `METHOD="POST"` is the following:

```
#include <stdio.h>
#include <stdlib.h>
#define MAXLEN 80
#define EXTRA 5
/* 4 for field name "data", 1 for "=" */
#define MAXINPUT MAXLEN+EXTRA+2
/* 1 for added line break, 1 for trailing NUL */
#define DATAFILE "../data/data.txt"
void unencode(char *src, char *last, char *dest)
{
    for(; src != last; src++, dest++)
        if(*src == '+')
            *dest = ' ';
        else if(*src == '%') {
            int code;
            if(sscanf(src+1, "%2x", &code) != 1) code = '?';
            *dest = code;
            src += 2; }
        else
            *dest = *src;
        *dest = '\n';
        *++dest = '\0';
    }

int main(void)
{
    char *lenstr;
    char input[MAXINPUT], data[MAXINPUT];
```

```

long len;
printf("Content-Type:text/html\n\n");
printf("<TITLE>Response</TITLE>\n");
lenstr = getenv("CONTENT_LENGTH");
if(lenstr == NULL || sscanf(lenstr,"%ld",&len)!=1 || len > MAXLEN)
    printf("<P>Error in invocation - wrong FORM probably.");
else {
    FILE *f;
    fgets(input, len+1, stdin);
    unencode(input+EXTRA, input+len, data);
    f = fopen(DATAFILE, "a");
    if(f == NULL)
        printf("<P>Sorry, cannot store your data.");
    else
        fputs(data, f);
    fclose(f);
    printf("<P>Thank you! Your contribution has been stored.");
}
return 0;
}

```

Essentially, the program retrieves the information about the number of characters in the input from value of the CONTENT\_LENGTH environment variable. Then it unencodes (decodes) the data, since the data arrives in the specifically encoded format that was already mentioned. The program has been written for a form where the text input field has the name data (actually, just the length of the name matters here). For example, if the user types Hello there! then the data will be passed to the program encoded as data=Hello+there%21 (with space encoded as + and exclamation mark encoded as %21). The unencode routine in the program converts this back to the original format. After that, the data is appended to a file (with a fixed file name), as well as echoed back to the user. Having compiled the program I have saved it as formpost.exe into the directory for CGI scripts. Now a form like the following can be used for data submissions:

```

<FORM ACTION=http://localhost/cgi-bin/formpost.exe METHOD="POST">
<DIV>Your input (80 chars max.):<BR>
<INPUT NAME="data" SIZE="60" MAXLENGTH="80"><BR">
<INPUT TYPE="SUBMIT" VALUE="Send"></DIV>
</FORM>

```

## Sample Programs in perl

```
#!/C:/Perl64/bin/perl.exe
##
## printenv -- demo CGI program which just prints its environment
##

print "Content-type: text/html", "\n\n";
print "<HTML>", "\n";
print "<HEAD><TITLE>About this Server</TITLE></HEAD>", "\n";
print "<BODY><H1>About this Server</H1>", "\n";
print "<HR><PRE>";
print "Server Name:   ", $ENV{'SERVER_NAME'}, "<BR>", "\n";
print "Running on Port: ", $ENV{'SERVER_PORT'}, "<BR>", "\n";
print "Server Software: ", $ENV{'SERVER_SOFTWARE'}, "<BR>", "\n";
print "Server Protocol: ", $ENV{'SERVER_PROTOCOL'}, "<BR>", "\n";
print "CGI Revision:   ", $ENV{'GATEWAY_INTERFACE'}, "<BR>", "\n";
print "<HR></PRE>", "\n";
print "</BODY></HTML>", "\n";
exit (0);
```

## program that displays environment variables in perl

```
#!/C:/Perl64/bin/perl.exe
##
## printenv -- demo CGI program which just prints its environment
##

print "Content-type: text/plain; charset=iso-8859-1\n\n";
foreach $var (sort(keys(%ENV))) {
    $val = $ENV{$var};
    $val =~ s/\n//g;
    $val =~ s/"/\"/g;
    print "${var}=\"${val}\"\\n";
}
```

## Sample Programs in C

```
#include<stdio.h>

main()
{
    printf("Content-Type:text/html\n\n");
}
```

```
printf("<HTML>\n");
printf("<HEAD> <TITLE>\n");
printf("This is a First CGI program \n");
printf("</TITLE>\n");
printf("<BODY>");
printf("This is first CGI program");
printf("</BODY>");
printf("</HTML>");
}
```

## **Module III**

### **3.1. Introduction to Java programming**

A new Object Oriented Programming Language developed at Sun Microsystems. Easier to learn than most other Object Oriented programming languages, since it has collected the best parts of the existing ones. A language that is standardized enough so that executable applications can run on any (in principle) computer that contains a Virtual Machine (run-time environment). Virtual machines can be embedded in web browsers (such as Netscape Navigator, Microsoft Internet Explorer, and IBM WebExplorer) and operating systems. A standardized set of Class Libraries (packages), that support creating graphical user interfaces, controlling multimedia data and communicating over networks. A programming language that supports the most recent features in computer science at the programming language level.

### **3.2. History of Java**

Java was conceived by James Gosling, Patrick Naughton, Chris Warth, Ed Frank, and Mike Sheridan at Sun Microsystems, Inc. in 1991. It took 18 months to develop the first working version. This language was initially called "Oak" but was renamed "Java" in 1995. Between the initial implementation of Oak in the fall of 1992 and the public announcement of Java in the spring of 1995, many more people contributed to the design and evolution of the language. Somewhat surprisingly, the original impetus for Java was not the Internet! Instead, the primary motivation was the need for a platform-independent language that could be used to create software to be embedded in various consumer electronic devices, such as toasters, microwave ovens, and remote controls. As you can probably guess, many different types of CPUs are used as controllers.

### **3.3. Features of Java**

No discussion of the genesis of Java is complete without a look at the Java features. Although the fundamental forces that necessitated the invention of Java are portability and security, other factors also played an important role in molding the final form of the language. The key considerations were summed up by the Java team in the following list of buzzwords:

- Simple
- Secure
- Portable
- Object-oriented
- Robust
- Multithreaded
- Architecture-neutral
- Interpreted
- High performance



- Distributed
- Dynamic

Two of these buzzwords have already been discussed: secure and portable. Let's examine what each of the others implies.

### **3.3.1. Simple**

Java was designed to be easy for the professional programmer to learn and use effectively. Assuming that you have some programming experience, you will not find Java hard to master. If you already understand the basic concepts of object-oriented programming, learning Java will be even easier. Best of all, if you are an experienced C++ programmer, moving to Java will require very little effort. Because Java inherits the C/C++ syntax and many of the object-oriented features of C++, most programmers have little trouble learning Java.

### **3.3.2. Security**

As you are likely aware, every time that you download a "normal" program, you are risking a viral infection. Prior to Java, most users did not download executable programs frequently, and those who did scanned them for viruses prior to execution. Even so, most users still worried about the possibility of infecting their systems with a virus. In addition to viruses, another type of malicious program exists that must be guarded against. This type of program can gather private information, such as credit card numbers, bank account balances, and passwords, by searching the contents of your computer's local file system. Java answers both of these concerns by providing a "firewall" between a networked application and your computer.

When you use a Java-compatible Web browser, you can safely download Java applets without fear of viral infection or malicious intent. Java achieves this protection by confining a Java program to the Java execution environment and not allowing it access to other parts of the computer. The ability to download applets with confidence that no harm will be done and that no security will be breached is considered by many to be the single most important aspect of Java.

### **3.3.3. Portability**

As discussed earlier, many types of computers and operating systems are in use throughout the world—and many are connected to the Internet. For programs to be dynamically downloaded to all the various types of platforms connected to the Internet, some means of generating portable executable code is needed. As you will soon see, the same mechanism that helps ensure security also helps create portability. Indeed, Java's solution to these two problems is both elegant and efficient.

### **3.3.4. Object-Oriented**

At the center of Java is object-oriented programming (OOP). The object-oriented methodology is inseparable from Java, and all Java programs are, to at least some extent, object-oriented. Because

of OOP's importance to Java, it is useful to understand OOP's basic principles before you write even a simple Java program.

### **3.3.5. Robust**

The ability to create robust programs was given a high priority in the design of Java. To gain reliability, Java restricts you in a few key areas, to force you to find your mistakes early in program development. At the same time, Java frees you from having to worry about many of the most common causes of programming errors. Because Java is a strictly typed language, it checks your code at compile time. However, it also checks your code at run time.

To better understand how Java is robust, consider two of the main reasons for program failure: memory management mistakes and mishandled exceptional conditions (that is, run-time errors). Memory management can be a difficult, tedious task in traditional programming environments. For example, in C/C++, the programmer must manually allocate and free all dynamic memory. This sometimes leads to problems, because programmers will either forget to free memory that has been previously allocated or, worse, try to free some memory that another part of their code is still using. Java virtually eliminates these problems by managing memory allocation and deallocation for you. (In fact, deallocation is completely automatic, because Java provides garbage collection for unused objects.) Exceptional conditions in traditional environments often arise in situations such as division by zero or "file not found," and they must be managed with clumsy and hard-to-read constructs. Java helps in this area by providing object-oriented exception handling. In a well-written Java program, all run-time errors can—and should—be managed by your program.

### **3.3.6. Multithreaded**

Java was designed to meet the real-world requirement of creating interactive, networked programs. To accomplish this, Java supports multithreaded programming, which allows you to write programs that do many things simultaneously. The Java run-time system comes with an elegant yet sophisticated solution for multiprocess synchronization that enables you to construct smoothly running interactive systems. Java's easy-to-use approach to multithreading allows you to think about the specific behavior of your program, not the multitasking subsystem.

### **3.3.7. Architecture-Neutral**

A central issue for the Java designers was that of code longevity and portability. One of the main problems facing programmers is that no guarantee exists that if you write a program today, it will run tomorrow—even on the same machine. Operating system upgrades, processor upgrades, and changes in core system resources can all combine to make a program malfunction. The Java designers made several hard decisions in the Java language and the Java Virtual Machine in an attempt to alter this situation. Their goal was "write once; run anywhere, any time, forever." To a great extent, this goal was accomplished.

### **3.3.8. Interpreted and High Performance**

As described earlier, Java enables the creation of cross-platform programs by compiling into an intermediate representation called Java bytecode. This code can be interpreted on any system that provides a Java Virtual Machine. Most previous attempts at cross-platform solutions have done so at the expense of performance. Other interpreted systems, such as BASIC, Tcl, and PERL, suffer from almost insurmountable performance deficits. Java, however, was designed to perform well on very low-power CPUs. As explained earlier, while it is true that Java was engineered for interpretation, the Java bytecode was carefully designed so that it would be easy to translate directly into native machine code for very high performance by using a just-in-time compiler. Java run-time systems that provide this feature lose none of the benefits of the platform-independent code. "High-performance cross-platform" is no longer an oxymoron.

### **3.3.9. Distributed**

Java is designed for the distributed environment of the Internet, because it handles TCP/IP protocols. In fact, accessing a resource using a URL is not much different from accessing a file. The original version of Java (Oak) included features for intra-address-space messaging. This allowed objects on two different computers to execute procedures remotely. Java has recently revived these interfaces in a package called Remote Method Invocation (RMI). This feature brings an unparalleled level of abstraction to client/server programming.

### **3.3.10. Dynamic**

Java programs carry with them substantial amounts of run-time type information that is used to verify and resolve accesses to objects at run time. This makes it possible to dynamically link code in a safe and expedient manner. This is crucial to the robustness of the applet environment, in which small fragments of bytecode may be dynamically updated on a running system.

## **3.4. Java Applets and Applications**

Java can be used to create two types of programs: applications and applets. An application is a program that runs on your computer, under the operating system of that computer. That is, an application created by Java is more or less like one created using C or C++. When used to create applications, Java is not much different from any other computer language. Rather, it is Java's ability to create applets that makes it important. An applet is an application designed to be transmitted over the Internet and executed by a Java-compatible Web browser. An applet is actually a tiny Java program, dynamically downloaded across the network, just like an image, sound file, or video clip. The important difference is that an applet is an intelligent program, not just an animation or media file. In other words, an applet is a program that can react to user input and dynamically change—not just run the same animation or sound over and over. As exciting as applets are, they would be nothing more than wishful thinking if Java were not able to address the two fundamental

problems associated with them: security and portability. Before continuing, let's define what these two terms mean relative to the Internet.

#### Application example

```
/*  
  
    This is a simple Java program.  
    Call this file Example.java.  
*/  
  
class Example {  
    // A Java program begins with a call to main().  
    public static void main(String args[]) {  
        System.out.println("Java drives the Web.");  
    }  
}  
  
Applet example  
/ A minimal applet.  
import java.awt.*;  
import java.applet.*;  
public class SimpleApplet extends Applet {  
    public void paint(Graphics g) {  
        g.drawString("Java makes applets easy.", 20, 20);  
    }  
}
```

### 3.5. The Java Development Kit (JDK)

The Java Development Kit (JDK) is the minimal file you need to download in order to develop in Java. The version that is meant for the Windows environment contains an automatic installer that takes care of the installation. Only setting up the Windows PATH may require some manual intervention. The JDK contains everything you need to develop general-purpose programs in Java:

#### Base Tools

*javac*: The Java Language Compiler that you use to compile programs written in the Java(tm) Programming Language into bytecodes.

*java*: The Java Interpreter that you use to run programs written in the Java(tm) Programming Language.

*appletviewer*: Allows you to run applets without a web browser.etc.

### 3.6. Building blocks

Java uses *unicode character set*. *Unicode* is computing industry standards designed to consistently and uniquely encode characters used in written languages throughout the world. The Unicode standard uses hexadecimal to express a character. For example, the value 0x0040 represents the Latin character A. The Unicode standard was initially designed using 16 bits to encode characters because the primary machines were 16-bit PCs.

When the specification for the Java language was created, the Unicode standard was accepted and the char primitive was defined as a 16-bit data type, with characters in the hexadecimal range from 0x0000 to 0xFFFF.

Because 16-bit encoding supports  $2^{16}$  (65,536) characters, which is insufficient to define all characters in use throughout the world, the Unicode standard was extended to 0x10FFFF, which supports over one million characters. The definition of a character in the Java programming language could not be changed from 16 bits to 32 bits without causing millions of Java applications to no longer run properly. To correct the definition, a scheme was developed to handle characters that could not be encoded in 16 bits.

### 3.7 Data types

Java defines eight simple (or elemental) types of data: byte, short, int, long, char, float, double, and boolean. These can be put in four groups:

- Integers This group includes byte, short, int, and long, which are for whole-valued signed numbers.
- Floating-point numbers This group includes float and double, which represent numbers with fractional precision.
- Characters This group includes char, which represents symbols in a character set, like letters and numbers.
- Boolean This group includes boolean, which is a special type for representing true/false values.

You can use these types as-is, or to construct arrays or your own class types. Thus, they form the basis for all other types of data that you can create.

#### 3.7.1. Integers

Java defines four integer types: byte, short, int, and long. All of these are signed, positive and negative values. Java does not support unsigned, positive-only integers. However, Java's designers felt that unsigned integers were unnecessary.

The width of an integer type should not be thought of as the amount of storage it consumes, but rather as the behavior it defines for variables and expressions of that type. The Java run-time environment is free to use whatever size it wants, as long as the types behave as you declared them. In fact, at least one implementation stores bytes and shorts as 32-bit (rather than 8- and 16-bit)

values to improve performance, because that is the word size of most computers currently in use. The width and ranges of these integer types vary widely, as shown in this table:

| Name | Width | Range |
|------|-------|-------|
|------|-------|-------|

|      |    |   |
|------|----|---|
| long | 64 | −9,223,372,036,854,775,808 to 9,223,372,036,854,775,807 |
|------|----|---|

|     |    |                                 |
|-----|----|---------------------------------|
| int | 32 | −2,147,483,648 to 2,147,483,647 |
|-----|----|---------------------------------|

|       |    |                   |
|-------|----|-------------------|
| short | 16 | −32,768 to 32,767 |
|-------|----|-------------------|

|      |   |             |
|------|---|-------------|
| byte | 8 | −128 to 127 |
|------|---|-------------|

Example

```
class Light {  
    public static void main(String args[]) {  
        int lightspeed;  
        long days;  
        long seconds;  
        long distance;  
        // approximate speed of light in miles per second  
        lightspeed = 186000;  
        days = 1000; // specify number of days here  
        seconds = days * 24 * 60 * 60; // convert to seconds  
        distance = lightspeed * seconds; // compute distance  
        System.out.print("In " + days);  
        System.out.print(" days light will travel about ");  
        System.out.println(distance + " miles.");  
    }  
}
```

This program generates the following output:

In 1000 days light will travel about 160704000000000 miles.

### 3.7.2. Floating-Point Types

Floating-point numbers, also known as real numbers, are used when evaluating expressions that require fractional precision. For example, calculations such as square root, or transcendental functions such as sine and cosine, result in a value whose precision requires a floating-point type. Java implements the standard (IEEE-754) set of floating-point types and operators. There are two kinds of floating-point types, float and double, which represent single- and double-precision numbers, respectively. Their width and ranges are shown here:

|        |    |                      |
|--------|----|----------------------|
| double | 64 | 1.7e−308 to 1.7e+308 |
|--------|----|----------------------|

|       |    |                      |
|-------|----|----------------------|
| float | 32 | 3.4e−038 to 3.4e+038 |
|-------|----|----------------------|

## **float**

The type float specifies a single-precision value that uses 32 bits of storage. Single precision is faster on some processors and takes half as much space as double precision, but will become imprecise when the values are either very large or very small.

## **double**

Double precision, as denoted by the double keyword, uses 64 bits to store a value. Double precision is actually faster than single precision on some modern processors that have been optimized for high-speed mathematical calculations. All transcendental math functions, such as `sin( )`, `cos( )`, and `sqrt( )`, return double values. When you need to maintain accuracy over many iterative calculations, or are manipulating large-valued numbers, double is the best choice.

Example: Here is a short program that uses double variables to compute the area of a circle:

```
// Compute the area of a circle.
class Area {
    public static void main(String args[]) {
        double pi, r, a;
        r = 10.8; // radius of circle
        pi = 3.1416; // pi, approximately
        a = pi * r * r; // compute area
        System.out.println("Area of circle is " + a);
    }
}
```

### **3.7.3. Characters**

In Java, the data type used to store characters is char. In C/C++, char is an integer type that is 8 bits wide. This is not the case in Java. Instead, Java uses Unicode to represent characters. Unicode defines a fully international character set that can represent all of the characters found in all human languages. It is a unification of dozens of character sets, such as Latin, Greek, Arabic, Cyrillic, Hebrew, Katakana, Hangul, and many more. For this purpose, it requires 16 bits. Thus, in Java char is a 16-bit type. The range of a char is 0 to 65,536. There are no negative chars.

Here is a program that demonstrates char variables:

```
// Demonstrate char data type.
class CharDemo {
    public static void main(String args[]) {
        char ch1, ch2;
        ch1 = 88; // code for X
        ch2 = 'Y';
    }
}
```

```

    System.out.print("ch1 and ch2: ");
    System.out.println(ch1 + " " + ch2);
}
}

```

This program displays the following output:

ch1 and ch2: X Y

#### 3.7.4. Booleans

Java has a simple type, called boolean, for logical values. It can have only one of two possible values, true or false. This is the type returned by all relational operators, such as  $a < b$ . boolean is also the type required by the conditional expressions that govern the control statements such as if and for. Here is a program that demonstrates the boolean type:

```

// Demonstrate boolean values.
class BoolTest {
    public static void main(String args[]) {
        boolean b;
        b = false;
        System.out.println("b is " + b);
        b = true;
        System.out.println("b is " + b);
        // a boolean value can control the if statement
        if(b) System.out.println("This is executed.");
        b = false;
        if(b) System.out.println("This is not executed.");
    }
}

```

### 3.8. Variables

The variable is the basic unit of storage in a Java program. A variable is defined by the combination of an identifier, a type, and an optional initializer. In addition, all variables have a scope, which defines their visibility, and a lifetime. These elements are examined next.

#### 3.8.1. Declaring a Variable

In Java, all variables must be declared before they can be used. The basic form of a variable declaration is shown here:

```

type identifier [= value][, identifier [= value] ...] ;

```

The type is one of Java's atomic types, or the name of a class or interface. The identifier is the name of the variable. You can initialize the variable by specifying an equal sign and a value. Keep in



mind that the initialization expression must result in a value of the same (or compatible) type as that specified for the variable. To declare more than one variable of the specified type, use a comma-separated list. Here are several examples of variable declarations of various types. Note that some include an initialization.

```
int a, b, c;
```

```
int d = 3, e, f = 5;
```

```
byte z = 22;
```

```
double pi = 3.14159;
```

```
char x = 'x';
```

### **3.8.2. The Scope and Lifetime of Variables**

So far, all of the variables used have been declared at the start of the `main( )` method. However, Java allows variables to be declared within any block. A block defines a scope. Thus, each time you start a new block, you are creating a new scope. As you probably know from your previous programming experience, a scope determines what objects are visible to other parts of your program. It also determines the lifetime of those objects.

Most other computer languages define two general categories of scopes: global and local. However, these traditional scopes do not fit well with Java's strict, object-oriented model. While it is possible to create what amounts to being a global scope, it is by far the exception, not the rule. In Java, the two major scopes are those defined by a class and those defined by a method. However, since the class scope has several unique properties and attributes that do not apply to the scope defined by a method, this distinction makes some sense. The scope defined by a method begins with its opening curly brace. However, if that method has parameters, they too are included within the method's scope. As a general rule, variables declared inside a scope are not visible (that is, accessible) to code that is defined outside that scope. Thus, when you declare a variable within a scope, you are localizing that variable and protecting it from unauthorized access and/or modification.

### **3.9. Type Conversion and Casting**

If you have previous programming experience, then you already know that it is fairly common to assign a value of one type to a variable of another type. If the two types are compatible, then Java will perform the conversion automatically. For example, it is always possible to assign an `int` value to a `long` variable. However, not all types are compatible, and thus, not all type conversions are implicitly allowed. For instance, there is no conversion defined from `double` to `byte`. Fortunately, it is still possible to obtain a conversion between incompatible types. To do so, you must use a cast, which performs an explicit conversion between incompatible types. Let's look at both automatic type conversions and casting.

#### **3.9.1. Java's Automatic Conversions**

When one type of data is assigned to another type of variable, an automatic type conversion will take place if the following two conditions are met:

1. The two types are compatible.
2. The destination type is larger than the source type.

When these two conditions are met, a widening conversion takes place. For example, the `int` type is always large enough to hold all valid `byte` values, so no explicit cast statement is required. For widening conversions, the numeric types, including integer and floating-point types, are compatible with each other. However, the numeric types are not compatible with `char` or `boolean`. Also, `char` and `boolean` are not compatible with each other.

### 3.9.2. Casting Incompatible Types

Although the automatic type conversions are helpful, they will not fulfill all needs. For example, what if you want to assign an `int` value to a `byte` variable? This conversion will not be performed automatically, because a `byte` is smaller than an `int`. This kind of conversion is sometimes called a narrowing conversion, since you are explicitly making the value narrower so that it will fit into the target type. To create a conversion between two incompatible types, you must use a cast. A cast is simply an explicit type conversion. It has this general form:

*(target-type) value*

Here, *target-type* specifies the desired type to convert the specified value to. Foreexample, the following fragment casts an `int` to a `byte`. If the integer's value is larger than the range of a `byte`, it will be reduced modulo (the remainder of an integer division by the) `byte`'s range.

```
int a;  
byte b;  
b = (byte) a;
```

A different type of conversion will occur when a floating-point value is assigned to an integer type: truncation. As you know, integers do not have fractional components. Thus, when a floating-point value is assigned to an integer type, the fractional component is lost.

### 3.10. Wrapper Classes

Each of Java's eight primitive data types has a class dedicated to it. These are known as wrapper classes, because they "wrap" the primitive data type into an object of that class. So, there is an `Integer` class that holds an `int` variable, there is a `Double` class that holds a `double` variable, and so on. The wrapper classes are part of the `java.lang` package, which is imported by default into all Java programs. The following discussion focuses on the `Integer` wrapper class, but applies in a general sense to all eight wrapper classes. Consult the Java API documentation for more details. The following two statements illustrate the difference between a primitive data type and an object of a wrapper class:

```
int
```

```
x = 25;
```

```
Integer y = new Integer(33);
```

The first statement declares an int variable named x and initializes it with the value 25. The second statement instantiates an Integer object. The object is initialized with the value 33 and a reference to the object is assigned to the object variable y.

Clearly x and y differ by more than their values: x is a variable that holds a value; y is an object variable that holds a reference to an object. As noted earlier, data fields in objects are not, in general, directly accessible. So, the following statement using x and y as declared above is not allowed:

```
int z = x + y; // wrong!
```

The data field in an Integer object is only accessible using the methods of the Integer class. One such method — the intValue() method — returns an int equal to the value of the object, effectively "unwrapping" the Integer object:

```
int z = x + y.intValue(); // OK!
```

Some of Java's classes include only instance methods, some include only class methods, some include both. The Integer class includes both instance methods and class methods. The class methods provide useful services; however, they are not called through instances of the class. But, this is not new. We met a class method of the Integer class earlier. The statement

```
int x = Integer.parseInt("1234");
```

converts the string "1234" into the integer 1,234 and assigns the result to the int variable x. The method parseInt() is a class method.

### 3.11. Operators

Java provides a rich operator environment. Most of its operators can be divided into the following four groups: arithmetic, bitwise, relational, and logical. Java also defines some additional operators that handle certain special situations.

#### 3.11.1. Arithmetic Operators

Arithmetic operators are used in mathematical expressions in the same way that they are used in algebra. The following table lists the arithmetic operators:

|    |                                  |
|----|----------------------------------|
| +  | - Addition                       |
| -  | - Subtraction (also unary minus) |
| *  | - Multiplication                 |
| /  | - Division                       |
| %  | - Modulus                        |
| ++ | - Increment                      |

- `+=` - Addition assignment
- `-=` - Subtraction assignment
- `*=` - Multiplication assignment
- `/=` - Division assignment
- `%=` - Modulus assignment
- `--` - Decrement

### The Modulus Operator

The modulus operator, `%`, returns the remainder of a division operation. It can be applied to floating-point types as well as integer types. (This differs from C/C++, in which the `%` can only be applied to integer types.) The following example program demonstrates the `%`:

```
// Demonstrate the % operator.

class Modulus {
    public static void main(String args[]) {
        int x = 42;
        double y = 42.3;
        System.out.println("x mod 10 = " + x % 10);
        System.out.println("y mod 10 = " + y % 10);
    }
}
```

### 3.11.2. The Bitwise Operators

Java defines several bitwise operators which can be applied to the integer types, `long`, `int`, `short`, `char`, and `byte`. These operators act upon the individual bits of their operands. They are summarized in the following table:

Operator Result

- `~` Bitwise unary NOT
- `&` Bitwise AND
- `|` Bitwise OR
- `^` Bitwise exclusive OR
- `>>` Shift right
- `>>>` Shift right zero fill
- `<<` Shift left
- `&=` Bitwise AND assignment
- `|=` Bitwise OR assignment
- `^=` Bitwise exclusive OR assignment
- `>>=` Shift right assignment

`>>>=` Shift right zero fill assignment

`<<=` Shift left assignment

Since the bitwise operators manipulate the bits within an integer, it is important to understand what effects such manipulations may have on a value. Specifically, it is useful to know how Java stores integer values and how it represents negative numbers. So, before continuing, let's briefly review these two topics.

### **The Bitwise AND**

The AND operator, `&`, produces a 1 bit if both operands are also 1. A zero is produced in all other cases. Here is an example:

### **The Bitwise OR**

The OR operator, `|`, combines bits such that if either of the bits in the operands is a 1, then the resultant bit is a 1, as shown here:

### **The Bitwise XOR**

The XOR operator, `^`, combines bits such that if exactly one operand is 1, then the result is 1. Otherwise, the result is zero. The following example shows the effect of the `^`. This example also demonstrates a useful attribute of the XOR operation. Notice how the bit pattern of 42 is inverted wherever the second operand has a 1 bit. Wherever the second operand has a 0 bit, the first operand is unchanged. You will find this property useful when performing some types of bit manipulations.

### **The Left Shift**

The left shift operator, `<<`, shifts all of the bits in a value to the left a specified number of times. It has this general form:

*value << num*

Shifting  $n$  positions towards left means multiplying the number by  $2^n$ .

### **The Right Shift**

The right shift operator, `>>`, shifts all of the bits in a value to the right a specified number of times. Its general form is shown here:

*value >> num*

Here, `num` specifies the number of positions to right-shift the value in `value`. That is, the `>>` moves all of the bits in the specified value to the right the number of bit positions specified by `num`. The following code fragment shifts the value 32 to the right by two positions, resulting in a being set to 8: shifting  $n$  positions towards right means dividing the number by  $2^n$ .

### **3.11.3. Relational Operators**

The relational operators determine the relationship that one operand has to the other. Specifically, they determine equality and ordering. The relational operators are shown here:

`==` Equal to

|                    |                          |
|--------------------|--------------------------|
| <code>!=</code>    | Not equal to             |
| <code>&gt;</code>  | Greater than             |
| <code>&lt;</code>  | Less than                |
| <code>&gt;=</code> | Greater than or equal to |
| <code>&lt;=</code> | Less than or equal to    |

The outcome of these operations is a boolean value.

#### 3.11.4. Boolean Logical Operators

The Boolean logical operators shown here operate only on boolean operands. All of the binary logical operators combine two boolean values to form a resultant boolean value.

|                         |                            |
|-------------------------|----------------------------|
| <code>&amp;</code>      | Logical AND                |
| <code> </code>          | Logical OR                 |
| <code>^</code>          | Logical XOR (exclusive OR) |
| <code>  </code>         | Short-circuit OR           |
| <code>&amp;&amp;</code> | Short-circuit AND          |
| <code>!</code>          | Logical unary NOT          |
| <code>&amp;=</code>     | AND assignment             |
| <code> =</code>         | OR assignment              |
| <code>^=</code>         | XOR assignment             |
| <code>==</code>         | Equal to                   |
| <code>!=</code>         | Not equal to               |

The logical Boolean operators, `&`, `|`, and `^`, operate on boolean values in the same way that they operate on the bits of an integer.

#### 3.11.5. The Assignment Operator

You have been using the assignment operator since Chapter 2. Now it is time to take a formal look at it. The assignment operator is the single equal sign, `=`. The assignment operator works in Java much as it does in any other computer language. It has this general form:

*var = expression;*

Here, the type of `var` must be compatible with the type of `expression`. The assignment operator does have one interesting attribute that you may not be familiar with: it allows you to create a chain of assignments. For example, consider this fragment:

#### 3.11.6. The `?:` Operator

Java includes a special ternary (three-way) operator that can replace certain types of if- then-else statements. This operator is the `?`, and it works in Java much like it does in C and C++. It can seem somewhat confusing at first, but the `?` can be used very effectively once mastered. The `?` has this general form:

*expression1 ? expression2 : expression3*

Here, expression1 can be any expression that evaluates to a boolean value. If expression1 is true, then expression2 is evaluated; otherwise, expression3 is evaluated. The result of the ? operation is that of the expression evaluated. Both expression2 and expression3 are required to return the same type, which can't be void.

### 3.12. Control structures

Flow controls which have an identical syntax to that of "C" are:

- if-else
- switch-case
- while
- do-while
- for
- break, continue, return

But there is no “goto” statement! Instead, Java proposes labelled breaks, as shown in the following example.

```
public class LabelledBreak
{
    public static void main(String argv[])
    {
        brkpnt: for ( int i = 0 ; i < 10 ; i++ ) {
            for ( int j = 0 ; j < 10 ; j++ ) {
                if ( j == 3 ) break brkpnt;
                System.out.println("i = " + i + " , j = " + j);
            }
        }
    }
}
```

If we would have an ordinary “break;” in Example 6, then we would go through the outer loop 10 times and through the inner one 3 times for each outer loop repetition. However, the labelled break breaks the outer loop straight away and we do not even finish the first repetition of the outer loop.

### 3.13. Arrays

An array is a group of like-typed variables that are referred to by a common name. Arrays of any type can be created and may have one or more dimensions. A specific element in an array is accessed by its index. Arrays offer a convenient means of grouping related information. Arrays in Java work differently than they do in C and C++.

### 3.13.1. One-Dimensional Arrays

A one-dimensional array is, essentially, a list of like-typed variables. To create an array, you first must create an array variable of the desired type. The general form of a one-dimensional array declaration is

```
type var-nam[ ];
```

Here, *type* declares the base type of the array. The base type determines the data type of each element that comprises the array. Thus, the base type for the array determines what type of data the array will hold. For example, the following declares an array named `month_days` with the type "array of int":

```
int month_days[ ];
```

Although this declaration establishes the fact that `month_days` is an array variable, no array actually exists. In fact, the value of `month_days` is set to null, which represents an array with no value. To link `month_days` with an actual, physical array of integers, you must allocate one using `new` and assign it to `month_days`. `new` is a special operator that allocates memory.

```
array-var = new type[size];
```

Here, *type* specifies the type of data being allocated, *size* specifies the number of elements in the array, and *array-var* is the array variable that is linked to the array. That is, to use `new` to allocate an array, you must specify the type and number of elements to allocate. The elements in the array allocated by `new` will automatically be initialized to zero. This example allocates a 12-element array of integers and links them to `month_days`.

```
month_days = new int[12];
```

After this statement executes, `month_days` will refer to an array of 12 integers. Further, all elements in the array will be initialized to zero.

Let's review: Obtaining an array is a two-step process. First, you must declare a variable of the desired array type. Second, you must allocate the memory that will hold the array, using `new`, and assign it to the array variable. Thus, in Java all arrays are dynamically allocated.

Arrays can be initialized when they are declared. The process is much the same as that used to initialize the simple types. An array initializer is a list of comma-separated expressions surrounded by curly braces. The commas separate the values of the array elements. The array will automatically be created large enough to hold the number of elements you specify in the array initializer. There is no need to use `new`.

```
int month_days[] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
```

### 3.13.2. Multidimensional Arrays

In Java, multidimensional arrays are actually arrays of arrays. These, as you might expect, look and act like regular multidimensional arrays. However, as you will see, there are a couple of subtle



differences. To declare a multidimensional array variable, specify each additional index using another set of square brackets. For example, the following declares a two-dimensional array variable called twoD.

```
int twoD[][] = new int[4][5];
```

This allocates a 4 by 5 array and assigns it to twoD. Internally this matrix is implemented as an array of arrays of int.

### **3.13.3. Alternative Array Declaration Syntax**

There is a second form that may be used to declare an array:

```
type[ ] var-name;
```

Here, the square brackets follow the type specifier, and not the name of the array variable. For example, the following two declarations are equivalent:

```
int a1[] = new int[3];
```

```
int[] a2 = new int[3];
```

The following declarations are also equivalent:

```
char twod1[][] = new char[3][4];
```

```
char[][] twod2 = new char[3][4];
```

This alternative declaration form is included mostly as a convenience.

### **3.14. String Handling**

As is the case in most other programming languages, in Java a string is a sequence of characters. But, unlike many other languages that implement strings as character arrays, Java implements strings as objects of type String. Implementing strings as built-in objects allows Java to provide a full complement of features that make string handling convenient. For example, Java has methods to compare two strings, search for a substring, concatenate two strings, and change the case of letters within a string. Also, String objects can be constructed a number of ways, making it easy to obtain a string when needed. Somewhat unexpectedly, when you create a String object, you are creating a string that cannot be changed. That is, once a String object has been created, you cannot change the characters that comprise that string. At first, this may seem to be a serious restriction.

However, such is not the case. You can still perform all types of string operations. The difference is that each time you need an altered version of an existing string, a new String object is created that contains the modifications. The original string is left unchanged. This approach is used because fixed, immutable strings can be implemented more efficiently than changeable ones. For those cases in which a modifiable string is desired, there is a companion class to String called StringBuffer, whose objects contain strings that can be modified after they are created.

Both the String and StringBuffer classes are defined in java.lang. Thus, they are available to all programs automatically. Both are declared final, which means that neither of these classes may be

subclassed. This allows certain optimizations that increase performance to take place on common string operations. One last point: To say that the strings within objects of type `String` are unchangeable means that the contents of the `String` instance cannot be changed after it has been created. However, a variable declared as a `String` reference can be changed to point at some other `String` object at any time.

### 3.14.1. The String Constructors

The `String` class supports several constructors. To create an empty `String`, you call the default constructor. For example,

```
String s = new String();
```

will create an instance of `String` with no characters in it. Frequently, you will want to create strings that have initial values. The `String` class provides a variety of constructors to handle this. To create a `String` initialized by an array of characters, use the constructor shown here:

```
String(char chars[ ])
```

Here is an example:

```
char chars[] = { 'a', 'b', 'c' };  
String s = new String(chars);
```

*This constructor initializes s with the string "abc".*

You can specify a subrange of a character array as an initializer using the following constructor:

```
String(char chars[ ], int startIndex, int numChars)
```

Here, `startIndex` specifies the index at which the subrange begins, and `numChars` specifies the number of characters to use. Here is an example:

```
char chars[] = { 'a', 'b', 'c', 'd', 'e', 'f' };  
String s = new String(chars, 2, 3);
```

This initializes `s` with the characters `cde`.

You can construct a `String` object that contains the same character sequence as another `String` object using this constructor:

```
String(String strObj)
```

Here, `strObj` is a `String` object. Consider this example:

```
// Construct one String from another.  
class MakeString {  
    public static void main(String args[]) {  
        char c[] = { 'J', 'a', 'v', 'a' };  
        String s1 = new String(c);  
        String s2 = new String(s1);  
        System.out.println(s1);
```

```
        System.out.println(s2);
    }
}
```

The output from this program is as follows:

*Java*

*Java*

### **3.15. StringBuffer**

A string buffer implements a mutable sequence of characters. A string buffer is like a `String`, but can be modified. At any point in time it contains some particular sequence of characters, but the length and content of the sequence can be changed through certain method calls. `StringBuffer` is a peer class of `String` that provides much of the functionality of strings. As you know, `String` represents fixed-length, immutable character sequences. In contrast, `StringBuffer` represents growable and writeable character sequences. `StringBuffer` may have characters and substrings inserted in the middle or appended to the end. `StringBuffer` will automatically grow to make room for such additions and often has more characters preallocated than are actually needed, to allow room for growth. Java uses both classes heavily, but many programmers deal only with `String` and let Java manipulate `StringBuffers` behind the scenes by using the overloaded `+` operator.

#### **3.15.1. StringBuffer Constructors**

`StringBuffer` defines these three constructors:

```
StringBuffer( )
StringBuffer(int size)
StringBuffer(String str)
```

The default constructor (the one with no parameters) reserves room for 16 characters without reallocation. The second version accepts an integer argument that explicitly sets the size of the buffer. The third version accepts a `String` argument that sets the initial contents of the `StringBuffer` object and reserves room for 16 more characters without reallocation. `StringBuffer` allocates room for 16 additional characters when no specific buffer length is requested, because reallocation is a costly process in terms of time. Also, frequent reallocations can fragment memory. By allocating room for a few extra characters, `StringBuffer` reduces the number of reallocations that take place.

Java provides the `StringBuffer` and `String` classes, and the `String` class is used to manipulate character strings that cannot be changed. Simply stated, objects of type `String` are read only and immutable. The `StringBuffer` class is used to represent characters that can be modified.

### **3.16. Object-oriented programming**

Object-oriented programming is at the core of Java. In fact, all Java programs are object-oriented—this isn't an option the way that it is in C++, for example. OOP is so integral to Java that

you must understand its basic principles before you can write even simple Java programs. **Object-oriented programming (OOP)** is a programming paradigm using "objects" – usually instances of a class – consisting of data fields and methods together with their interactions – to design applications and computer programs. Programming techniques may include features such as data abstraction, encapsulation, messaging, modularity, polymorphism, and inheritance. Many modern programming languages now support OOP, at least as an option. As you know, all computer programs consist of two elements: code and data.

Furthermore, a program can be conceptually organized around its code or around its data. That is, some programs are written around "what is happening" and others are written around "who is being affected." These are the two paradigms that govern how a program is constructed. The first way is called the process-oriented model.

#### **3.16.1. Object**

An object is a software bundle of related state and behavior. Software objects are often used to model the real-world objects that you find in everyday life. This lesson explains how state and behavior are represented within an object, introduces the concept of data encapsulation, and explains the benefits of designing your software in this manner.

#### **3.16.2. Class**

A class is a blueprint or prototype from which objects are created. This section defines a class that models the state and behavior of a real-world object. It intentionally focuses on the basics, showing how even a simple class can cleanly model state and behavior.

#### **3.16.3. Inheritance**

Inheritance provides a powerful and natural mechanism for organizing and structuring your software. This section explains how classes inherit state and behavior from their superclasses, and explains how to derive one class from another using the simple syntax provided by the Java programming language.

#### **3.16.4. Interfaces**

An interface is a contract between a class and the outside world. When a class implements an interface, it promises to provide the behavior published by that interface. This section defines a simple interface and explains the necessary changes for any class that implements it.

#### **3.16.5. Packages**

A package is a namespace for organizing classes and interfaces in a logical manner. Placing your code into packages makes large software projects easier to manage. This section explains why this is useful, and introduces you to the Application Programming Interface (API) provided by the Java platform.

#### **3.16.6. Encapsulation**

Encapsulation is the mechanism that binds together code and the data it manipulates, and keeps both safe from outside interference and misuse. One way to think about encapsulation is as a protective wrapper that prevents the code and data from being arbitrarily accessed by other code defined outside the wrapper. Access to the code and data inside the wrapper is tightly controlled through a well-defined interface.

### **3.16.7. Polymorphism**

Polymorphism (from the Greek, meaning "many forms") is a feature that allows one interface to be used for a general class of actions. The specific action is determined by the exact nature of the situation.

### **3.17. Defining Classes**

When you define a class, you declare its exact form and nature. You do this by specifying the data that it contains and the code that operates on that data. While very simple classes may contain only code or only data, most real-world classes contain both. As you will see, a class' code defines the interface to its data. A class is declared by use of the class keyword. The classes that have been used up to this point are actually very limited examples of its complete form. Classes can (and usually do) get much more complex. The general form of a class definition is shown here:

```
class classname {
type instance-variable1;
type instance-variable2;
// ...
type instance-variableN;
type methodname1(parameter-list) {
// body of method
}
type methodname2(parameter-list) {
// body of method
}
// ...
type methodnameN(parameter-list) {
// body of method
}
}
```

The data, or variables, defined within a class are called instance variables. The code is contained within methods. Collectively, the methods and variables defined within a class are called members of the class. In most classes, the instance variables are acted upon and accessed by the methods

defined for that class. Thus, it is the methods that determine how a class' data can be used. Variables defined within a class are called instance variables because each instance of the class (that is, each object of the class) contains its own copy of these variables. Thus, the data for one object is separate and unique from the data for another. We will come back to this point shortly, but it is an important concept to learn early. All methods have the same general form as `main( )`, which we have been using thus far. However, most methods will not be specified as `static` or `public`. Notice that the general form of a class does not specify a `main( )` method. Java classes do not need to have a `main()` method. You only specify one if that class is the starting point for your program. Further, applets don't require a `main( )` method at all.

### **3.17.1. A Simple Class**

Let's begin our study of the class with a simple example. Here is a class called `Box` that defines three instance variables: `width`, `height`, and `depth`.

```
class Box {  
    double width;  
    double height;  
    double depth;  
}
```

As stated, a class defines a new type of data. In this case, the new data type is called `Box`. You will use this name to declare objects of type `Box`. It is important to remember that a class declaration only creates a template; it does not create an actual object. Thus, the preceding code does not cause any objects of type `Box` to come into existence.

To actually create a `Box` object, you will use a statement like the following: `Box mybox = new Box();` // create a `Box` object called `mybox` After this statement executes, `mybox` will be an instance of `Box`. Thus, it will have "physical" reality. For the moment, don't worry about the details of this statement.

Again, each time you create an instance of a class, you are creating an object that contains its own copy of each instance variable defined by the class. Thus, every `Box` object will contain its own copies of the instance variables `width`, `height`, and `depth`. To access these variables, you will use the dot (`.`) operator. The dot operator links the name of the object with the name of an instance variable. For example, to assign the `width` variable of `mybox` the value 100, you would use the following statement:

```
mybox.width = 100;
```

This statement tells the compiler to assign the copy of `width` that is contained within the `mybox` object the value of 100. In general, you use the dot operator to access both the instance variables and the methods within an object.

Here is a complete program that uses the Box class:

```
/* A program that uses the Box class.  
Call this file BoxDemo.java  
*/  
  
class Box {  
    double width;  
    double height;  
    double depth;  
}  
  
// This class declares an object of type Box.  
class BoxDemo {  
    public static void main(String args[]) {  
        Box mybox = new Box();  
        double vol;  
        // assign values to mybox's instance variables  
        mybox.width = 10;  
        mybox.height = 20;  
        mybox.depth = 15;  
        // compute volume of box  
        vol = mybox.width * mybox.height * mybox.depth;  
        System.out.println("Volume is " + vol);  
    }  
}
```

### **3.18. Access Modifiers In Java**

Access modifiers specifies who can access them. There are four access modifiers used in java. They are public, private, protected, no modifier (declaring without an access modifier). Using ‘no modifier’ is also sometimes referred as ‘package-private’ or ‘default’ or ‘friendly’ access. Usage of these access modifiers is restricted to two levels. The two levels are class level access modifiers and member level access modifiers.

#### **3.18.1. Class level access modifiers**

Only two access modifiers is allowed, *public* and *no modifier*

- If a class is ‘public’, then it CAN be accessed from ANYWHERE.
- If a class has ‘no modifier’, then it CAN ONLY be accessed from ‘same package’.

#### **3.18.2. Member level access modifiers (java variables and java methods)**

All the four *public*, *private*, *protected* and *no modifier* is allowed.

- public and no modifier – the same way as used in class level.
- private – members CAN ONLY access.
- protected – CAN be accessed from ‘same package’ and a subclass existing in any package can access.

For better understanding, member level access is formulated as a table:

| <i>Access Modifiers</i>   | <i>Same Class</i> | <i>Same Package</i> | <i>Subclass</i> | <i>Other packages</i> |
|---------------------------|-------------------|---------------------|-----------------|-----------------------|
| <b>public</b>             | Y                 | Y                   | Y               | Y                     |
| <b>protected</b>          | Y                 | Y                   | Y               | N                     |
| <b>no access modifier</b> | Y                 | Y                   | N               | N                     |
| <b>private</b>            | Y                 | N                   | N               | N                     |

First row {public Y Y Y Y} should be interpreted as:

- Y – A member declared with ‘public’ access modifier CAN be accessed by the members of the ‘same class’.
- Y – A member declared with ‘public’ access modifier CAN be accessed by the members of the ‘same package’.
- Y – A member declared with ‘public’ access modifier CAN be accessed by the members of the ‘subclass’.
- Y – A member declared as ‘public’ CAN be accessed from ‘Other packages’.

Second row {protected Y Y Y N} should be interpreted as:

- Y – A member declared with ‘protected’ access modifier CAN be accessed by the members of the ‘same class’.
- Y – A member declared with ‘protected’ access modifier CAN be accessed by the members of the ‘same package’.
- Y – A member declared with ‘protected’ access modifier CAN be accessed by the members of the ‘subclass’.
- N – A member declared with ‘protected’ access modifier CANNOT be accessed by the members of the ‘Other package’.

similarly interpret the access modifiers table for the third (no access modifier) and fourth (private access modifier) records.

### 3.19. Packages

Packages are class containers. In the preceding chapters, the name of each example class was taken from the same name space. This means that a unique name had to be used for each class to avoid name collisions. After a while, without some way to manage the name space, you could run out of



convenient, descriptive names for individual classes. You also need some way to be assured that the name you choose for a class will be reasonably unique and not collide with class names chosen by other programmers.

Thankfully, Java provides a mechanism for partitioning the class name space into more manageable chunks. This mechanism is the package. The package is both a naming and a visibility control mechanism. You can define classes inside a package that are not accessible by code outside that package. You can also define class members that are only exposed to other members of the same package. This allows your classes to have intimate knowledge of each other, but not expose that knowledge to the rest of the world.

### **3.19.1. Defining a Package**

To create a package is quite easy: simply include a package command as the first statement in a Java source file. Any classes declared within that file will belong to the specified package. The package statement defines a name space in which classes are stored. If you omit the package statement, the class names are put into the default package, which has no name. While the default package is fine for short, sample programs, it is inadequate for real applications. Most of the time, you will define a package for your code.

This is the general form of the package statement:

```
package pkg;
```

Here, *pkg* is the name of the package. For example, the following statement creates a package called *MyPackage*.

```
package MyPackage;
```

Java uses file system directories to store packages. For example, the .class files for any classes you declare to be part of *MyPackage* must be stored in a directory called *MyPackage*. Remember that case is significant, and the directory name must match the package name exactly.

More than one file can include the same package statement. The package statement simply specifies to which package the classes defined in a file belong. It does not exclude other classes in other files from being part of that same package. Most real-world packages are spread across many files.

You can create a hierarchy of packages. To do so, simply separate each package name from the one above it by use of a period. The general form of a multileveled package statement is shown here:

```
package pkg1[.pkg2[.pkg3]];
```

A package hierarchy must be reflected in the file system of your Java development system. For example, a package declared as

```
package java.awt.image;
```

needs to be stored in java/awt/image, java\\awt\\image, or java:awt:image on your UNIX, Windows, or Macintosh file system, respectively. Be sure to choose your package names carefully. You cannot rename a package without renaming the directory in which the classes are stored.

### 3.19.2. Understanding CLASSPATH

Before an example that uses a package is presented, a brief discussion of the CLASSPATH environmental variable is required. While packages solve many problems from an access control and name-space-collision perspective, they cause some curious difficulties when you compile and run programs. This is because the specific location that the Java compiler will consider as the root of any package hierarchy is controlled by CLASSPATH. Until now, you have been storing all of your classes in the same, unnamed default package. Doing so allowed you to simply compile the source code and run the Java interpreter on the result by naming the class on the command line. This worked because the default current working directory (.) is usually in the CLASSPATH environmental variable defined for the Java run-time system, by default. However, things are not so easy when packages are involved. Here's why.

### 3.19.3. A Short Package Example

Keeping the preceding discussion in mind, you can try this simple package:

```
// A simple package
package MyPack;

class Balance {
    String name;
    double bal;

    Balance(String n, double b) {
        name = n;
        bal = b;
    }

    void show() {
        if(bal<0)
            System.out.print("—> ");
        System.out.println(name + ": $" + bal);
    }

    class AccountBalance {
        public static void main(String args[]) {
            Balance current[] = new Balance[3];
            current[0] = new Balance("K. J. Fielding", 123.23);
```

```

    current[1] = new Balance("Will Tell", 157.02);
    current[2] = new Balance("Tom Jackson", -12.33);
    for(int i=0; i<3; i++) current[i].show();
}
}

```

Call this file AccountBalance.java, and put it in a directory called MyPack. Next, compile the file. Make sure that the resulting .class file is also in the MyPack directory. Then try executing the AccountBalance class, using the following command line:

```
java MyPack.AccountBalance
```

Remember, you will need to be in the directory above MyPack when you execute this command, or to have your CLASSPATH environmental variable set appropriately.

As explained, AccountBalance is now part of the package MyPack. This means that it cannot be executed by itself. That is, you cannot use this command line:

```
java AccountBalance
```

AccountBalance must be qualified with its package name.

### 3.20. Interfaces

Using the keyword interface, you can fully abstract a class' interface from its implementation. That is, using interface, you can specify what a class must do, but not how it does it. Interfaces are syntactically similar to classes, but they lack instance variables, and their methods are declared without any body. In practice, this means that you can define interfaces which don't make assumptions about how they are implemented. Once it is defined, any number of classes can implement an interface.

Also, one class can implement any number of interfaces. To implement an interface, a class must create the complete set of methods defined by the interface. However, each class is free to determine the details of its own implementation. By providing the interface keyword, Java allows you to fully utilize the "one interface, multiple methods" aspect of polymorphism. Interfaces are designed to support dynamic method resolution at run time.

#### 3.20.1. Defining an Interface

An interface is defined much like a class. This is the general form of an interface:

```

access interface name {
    return-type method-name1(parameter-list);
    return-type method-name2(parameter-list);
    type final-varname1 = value;
    type final-varname2 = value;
    // ...
}

```

```

    return-type method-nameN(parameter-list);
    type final-varnameN = value;
}

```

Here, access is either public or not used. When no access specifier is included, then default access results, and the interface is only available to other members of the package in which it is declared. When it is declared as public, the interface can be used by any other code. name is the name of the interface, and can be any valid identifier.

Notice that the methods which are declared have no bodies. They end with a semicolon after the parameter list. They are, essentially, abstract methods; there can be no default implementation of any method specified within an interface. Each class that includes an interface must implement all of the methods.

Variables can be declared inside of interface declarations. They are implicitly final and static, meaning they cannot be changed by the implementing class. They must also be initialized with a constant value. All methods and variables are implicitly public if the interface, itself, is declared as public.

Here is an example of an interface definition. It declares a simple interface which contains one method called callback( ) that takes a single integer parameter.

```

interface Callback {
    void callback(int param);
}

```

### 3.20.2. Implementing Interfaces

Once an interface has been defined, one or more classes can implement that interface. To implement an interface, include the implements clause in a class definition, and then create the methods defined by the interface. The general form of a class that includes the implements clause looks like this:

```

access class classname [extends superclass] [implements interface [,interface...]] {
    // class-body
}

```

Here, access is either public or not used. If a class implements more than one interface, the interfaces are separated with a comma. If a class implements two interfaces that declare the same method, then the same method will be used by clients of either interface. The methods that implement an interface must be declared public. Also, the type signature of the implementing method must match exactly the type signature specified in the interface definition.

Here is a small example class that implements the Callback interface shown earlier.

```

class Client implements Callback {

```

```
// Implement Callback's interface
public void callback(int p) {
    System.out.println("callback called with " + p);
}
}
```

Notice that `callback( )` is declared using the public access specifier. Note When you implement an interface method, it must be declared as public. It is both permissible and common for classes that implement interfaces to define additional members of their own. For example, the following version of `Client` implements `callback( )` and adds the method `nonIfaceMeth( )`:

```
class Client implements Callback {
// Implement Callback's interface
public void callback(int p) {
    System.out.println("callback called with " + p);
}
}
void nonIfaceMeth() {
    System.out.println("Classes that implement interfaces " +
        "may also define other members, too.");
}
```

### **3.21. Exception-Handling Fundamentals**

A Java exception is an object that describes an exceptional (that is, error) condition that has occurred in a piece of code. When an exceptional condition arises, an object representing that exception is created and thrown in the method that caused the error. That method may choose to handle the exception itself, or pass it on. Either way, at some point, the exception is caught and processed. Exceptions can be generated by the Java run-time system, or they can be manually generated by your code. Exceptions thrown by Java relate to fundamental errors that violate the rules of the Java language or the constraints of the Java execution environment. Manually generated exceptions are typically used to report some error condition to the caller of a method.

Java exception handling is managed via five keywords: `try`, `catch`, `throw`, `throws`, and `finally`. Briefly, here is how they work. Program statements that you want to monitor for exceptions are contained within a `try` block. If an exception occurs within the `try` block, it is thrown. Your code can catch this exception (using `catch`) and handle it in some rational manner. System-generated exceptions are automatically thrown by the Java run- time system. To manually throw an exception, use the keyword `throw`. Any exception that is thrown out of a method must be specified as such by a

throws clause. Any code that absolutely must be executed before a method returns is put in a finally block. This is the general form of an exception-handling block:

```
try {  
    // block of code to monitor for errors  
}  
catch (ExceptionType1 exOb) {  
    // exception handler for ExceptionType1  
}  
catch (ExceptionType2 exOb) {  
    // exception handler for ExceptionType2  
}  
// ...  
finally {  
    // block of code to be executed before try block ends  
}
```

### **3.21.1. Exception Types**

All exception types are subclasses of the built-in class Throwable. Thus, Throwable is at the top of the exception class hierarchy. Immediately below Throwable are two subclasses that partition exceptions into two distinct branches. One branch is headed by Exception. This class is used for exceptional conditions that user programs should catch. This is also the class that you will subclass to create your own custom exception types. There is an important subclass of Exception, called RuntimeException.

Exceptions of this type are automatically defined for the programs that you write and include things such as division by zero and invalid array indexing. The other branch is topped by Error, which defines exceptions that are not expected to be caught under normal circumstances by your program. Exceptions of type Error are used by the Java run-time system to indicate errors having to do with the run-time environment, itself. Stack overflow is an example of such an error.

### **3.21.2. Using try and catch**

Although the default exception handler provided by the Java run-time system is useful for debugging, you will usually want to handle an exception yourself. Doing so provides two benefits. First, it allows you to fix the error. Second, it prevents the program from automatically terminating. To guard against and handle a run-time error, simply enclose the code that you want to monitor inside a try block. Immediately following the try block, include a catch clause that specifies the exception type that you wish to catch. To illustrate how easily this can be done, the following

program includes a try block and a catch clause which processes the `ArithmeticException` generated by the division-by-zero error:

```
class Exc2 {  
    public static void main(String args[]) {  
        int d, a;  
        try { // monitor a block of code.  
            d = 0;  
            a = 42 / d;  
            System.out.println("This will not be printed.");  
        } catch (ArithmeticException e) { // catch divide-by-zero error  
            System.out.println("Division by zero.");  
        }  
        System.out.println("After catch statement.");  
    }  
}
```

This program generates the following output:

*Division by zero.*

*After catch statement.*

Notice that the call to `println( )` inside the try block is never executed. Once an exception is thrown, program control transfers out of the try block into the catch block. Put differently, catch is not "called," so execution never "returns" to the try block from a catch. Thus, the line "This will not be printed." is not displayed. Once the catch statement has executed, program control continues with the next line in the program following the entire try/catch mechanism.

### **3.21.3. Multiple catch Clauses**

In some cases, more than one exception could be raised by a single piece of code. To handle this type of situation, you can specify two or more catch clauses, each catching a different type of exception. When an exception is thrown, each catch statement is inspected in order, and the first one whose type matches that of the exception is executed. After one catch statement executes, the others are bypassed, and execution continues after the try/catch block. The following example traps two different exception types:

```
// Demonstrate multiple catch statements.  
class MultiCatch {  
    public static void main(String args[]) {  
        try {  
            int a = args.length;
```

```

System.out.println("a = " + a);
int b = 42 / a;
int c[] = { 1 };
c[42] = 99;
} catch(ArithmeticException e) {
System.out.println("Divide by 0: " + e);
} catch(ArrayIndexOutOfBoundsException e) {
System.out.println("Array index oob: " + e);
}
System.out.println("After try/catch blocks.");
}
}

```

This program will cause a division-by-zero exception if it is started with no command-line parameters, since a will equal zero. It will survive the division if you provide a command-line argument, setting a to something larger than zero. But it will cause an `ArrayIndexOutOfBoundsException`, since the int array c has a length of 1, yet the program attempts to assign a value to `c[42]`.

Here is the output generated by running it both ways:

```

C:\>java MultiCatch
a = 0
Divide by 0: java.lang.ArithmeticException: / by zero
After try/catch blocks.
C:\>java MultiCatch TestArg
a = 1
Array index oob: java.lang.ArrayIndexOutOfBoundsException: 42
After try/catch blocks.

```

When you use multiple catch statements, it is important to remember that exception subclasses must come before any of their superclasses. This is because a catch statement that uses a superclass will catch exceptions of that type plus any of its subclasses. Thus, a subclass would never be reached if it came after its superclass.

#### 3.21.4. Nested try Statements

The try statement can be nested. That is, a try statement can be inside the block of another try. Each time a try statement is entered, the context of that exception is pushed on the stack. If an inner try statement does not have a catch handler for a particular exception, the stack is unwound and the next try statement's catch handlers are inspected for a match. This continues until one of the catch



statements succeeds, or until all of the nested try statements are exhausted. If no catch statement matches, then the Java run-time system will handle the exception.

### 3.21.5. throw

So far, you have only been catching exceptions that are thrown by the Java run-time system. However, it is possible for your program to throw an exception explicitly, using the throw statement. The general form of throw is shown here:

```
throw ThrowableInstance;
```

Here, ThrowableInstance must be an object of type Throwable or a subclass of Throwable. Simple types, such as int or char, as well as non-Throwable classes, such as String and Object, cannot be used as exceptions. There are two ways you can obtain a Throwable object: using a parameter into a catch clause, or creating one with the new operator.

The flow of execution stops immediately after the throw statement; any subsequent statements are not executed. The nearest enclosing try block is inspected to see if it has a catch statement that matches the type of the exception. If it does find a match, control is transferred to that statement. If not, then the next enclosing try statement is inspected, and so on. If no matching catch is found, then the default exception handler halts the program and prints the stack trace.

Here is a sample program that creates and throws an exception. The handler that catches the exception rethrows it to the outer handler.

```
// Demonstrate throw.  
class ThrowDemo {  
    static void demoproc() {  
        try {  
            throw new NullPointerException("demo");  
        } catch(NullPointerException e) {  
            System.out.println("Caught inside demoproc.");  
            throw e; // rethrow the exception  
        }  
    }  
    public static void main(String args[]) {  
        try {  
            demoproc();  
        } catch(NullPointerException e) {  
            System.out.println("Recaught: " + e);  
        }  
    }  
}
```

### 3.21.6. throws

If a method is capable of causing an exception that it does not handle, it must specify this behavior so that callers of the method can guard themselves against that exception. You do this by including a throws clause in the method's declaration. A throws clause lists the types of exceptions that a method might throw. This is necessary for all exceptions, except those of type `Error` or `RuntimeException`, or any of their subclasses. All other exceptions that a method can throw must be declared in the throws clause. If they are not, a compile-time error will result.

This is the general form of a method declaration that includes a throws clause:

```
type method-name(parameter-list) throws exception-list
{
    // body of method
}
```

Here, exception-list is a comma-separated list of the exceptions that a method can throw. Following is an example of an incorrect program that tries to throw an exception that it does not catch. Because the program does not specify a throws clause to declare this fact, the program will not compile.

```
// This program contains an error and will not compile.
class ThrowsDemo {
    static void throwOne() {
        System.out.println("Inside throwOne.");
        throw new IllegalAccessException("demo");
    }
    public static void main(String args[]) {
        throwOne();
    }
}
```

To make this example compile, you need to make two changes. First, you need to declare that `throwOne()` throws `IllegalAccessException`. Second, `main()` must define a try/catch statement that catches this exception. The corrected example is shown here:

```
// This is now correct.
class ThrowsDemo {
    static void throwOne() throws IllegalAccessException {
        System.out.println("Inside throwOne.");
        throw new IllegalAccessException("demo");
    }
}
```

```

public static void main(String args[]) {
    try {
        throwOne();
    } catch (IllegalAccessException e) {
        System.out.println("Caught " + e);
    }
}
}
}

```

Here is the output generated by running this example program:

```

inside throwOne
caught java.lang.IllegalAccessException: demo

```

### 3.21.7. finally

When exceptions are thrown, execution in a method takes a rather abrupt, nonlinear path that alters the normal flow through the method. Depending upon how the method is coded, it is even possible for an exception to cause the method to return prematurely. This could be a problem in some methods. For example, if a method opens a file upon entry and closes it upon exit, then you will not want the code that closes the file to be bypassed by the exception-handling mechanism. The finally keyword is designed to address this contingency. finally creates a block of code that will be executed after a try/catch block has completed and before the code following the try/catch block. The finally block will execute whether or not an exception is thrown. If an exception is thrown, the finally block will execute even if no catch statement matches the exception. Any time a method is about to return to the caller from inside a try/catch block, via an uncaught exception or an explicit return statement, the finally clause is also executed just before the method returns. This can be useful for closing file handles and freeing up any other resources that might have been allocated at the beginning of a method with the intent of disposing of them before returning. The finally clause is optional. However, each try statement requires at least one catch or a finally clause.

Here is an example program that shows three methods that exit in various ways, none without executing their finally clauses:

```

// Demonstrate finally.
class FinallyDemo {
    // Through an exception out of the method.
    static void procA() {
        try {
            System.out.println("inside procA");
            throw new RuntimeException("demo");
        }
    }
}

```

```

    } finally {
        System.out.println("procA's finally");
    }
}

// Return from within a try block.
static void procB() {
    try {
        System.out.println("inside procB");
        return;
    } finally {
    }
}

System.out.println("procB's finally");
// Execute a try block normally.
static void procC() {
    try {
        System.out.println("inside procC");
    } finally {
        System.out.println("procC's finally");
    }
}

public static void main(String args[]) {
    try {
        procA();
    } catch (Exception e) {
        System.out.println("Exception caught");
    }
    procB();
    procC();
}
}

```

In this example, `procA()` prematurely breaks out of the try by throwing an exception. The finally clause is executed on the way out. `procB()`'s try statement is exited via a return statement. The finally clause is executed before `procB()` returns. In `procC()`, the try statement executes normally,

without error. However, the finally block is still executed. Note If a finally block is associated with a try, the finally block will be executed upon conclusion of the try.

Here is the output generated by the preceding program:

```
inside procA
procA's finally
Exception caught
inside procB
procB's finally
inside procC
procC's finally
```

### 3.21.8. Creating Your Own Exception Subclasses

Although Java's built-in exceptions handle most common errors, you will probably want to create your own exception types to handle situations specific to your applications. This is quite easy to do: just define a subclass of Exception (which is, of course, a subclass of Throwable). Your subclasses don't need to actually implement anything—it is their existence in the type system that allows you to use them as exceptions. The Exception class does not define any methods of its own. It does, of course, inherit those methods provided by Throwable. Thus, all exceptions, including those that you create, have the methods defined by Throwable available to them. The following example declares a new subclass of Exception and then uses that subclass to signal an error condition in a method. It overrides the toString( ) method, allowing the description of the exception to be displayed using println( ).

```
// This program creates a custom exception type.
class MyException extends Exception {
    private int detail;
    MyException(int a) {
        detail = a;
    }
    public String toString() {
        return "MyException[" + detail + "]";
    }
}
class ExceptionDemo {
    static void compute(int a) throws MyException {
        System.out.println("Called compute(" + a + ")");
        if(a > 10)
```

```

        throw new MyException(a);
        System.out.println("Normal exit");
    }
}

public static void main(String args[]) {
    try {
        compute(1);
        compute(20);
    } catch (MyException e) {
        System.out.println("Caught " + e);
    }
}

```

This example defines a subclass of Exception called MyException. This subclass is quite simple: it has only a constructor plus an overloaded toString( ) method that displays the value of the exception. The ExceptionDemo class defines a method named compute( ) that throws a MyException object. The exception is thrown when compute()'s integer parameter is greater than 10. The main( ) method sets up an exception handler for MyException, then calls compute( ) with a legal value (less than 10) and an illegal one to show both paths through the code. Here is the result:

### **3.21.9. Advantages of exception handling**

Exception handling provides a powerful mechanism for controlling complex programs that have many dynamic run-time characteristics. It is important to think of try, throw, and catch as clean ways to handle errors and unusual boundary conditions in your program's logic.

Exception handling helps us catch or identify abnormal scenarios in our code and handle them appropriately instead of throwing up a random error on the front-end (User Interface) of the application.

Exception handling allows developers to detect errors easily without writing special code to test return values. Even better, it lets us keep exception-handling code cleanly separated from the exception-generating code. It also lets us use the same exception-handling code to deal with a range of possible exceptions.

### **3.22. Multithreaded Programming**

Unlike most other computer languages, Java provides built-in support for multithreaded programming. A multithreaded program contains two or more parts that can run concurrently. Each part of such a program is called a thread, and each thread defines a separate path of execution. Thus, multithreading is a specialized form of multitasking. You are almost certainly acquainted with multitasking, because it is supported by virtually all modern operating systems. However, there are

two distinct types of multitasking: process-based and thread-based. It is important to understand the difference between the two. For most readers, process-based multitasking is the more familiar form. A process is, in essence, a program that is executing. Thus, process-based multitasking is the feature that allows your computer to run two or more programs concurrently. For example, process-based multitasking enables you to run the Java compiler at the same time that you are using a text editor. In process-based multitasking, a program is the smallest unit of code that can be dispatched by the scheduler. In a thread-based multitasking environment, the thread is the smallest unit of dispatchable code. This means that a single program can perform two or more tasks simultaneously. For instance, a text editor can format text at the same time that it is printing, as long as these two actions are being performed by two separate threads. Thus, process-based multitasking deals with the "big picture," and thread-based multitasking handles the details.

### **3.22.1. The Java Thread Model**

The Java run-time system depends on threads for many things, and all the class libraries are designed with multithreading in mind. In fact, Java uses threads to enable the entire environment to be asynchronous. This helps reduce inefficiency by preventing the waste of CPU cycles.

### **3.22.2. The Thread Class and the Runnable Interface**

Java's multithreading system is built upon the Thread class, its methods, and its companion interface, Runnable. Thread encapsulates a thread of execution. Since you can't directly refer to the ethereal state of a running thread, you will deal with it through its proxy, the Thread instance that spawned it. To create a new thread, your program will either extend Thread or implement the Runnable interface.

The Thread class defines several methods that help manage threads.

| <b>Method</b> | <b>Meaning</b> |
|---------------|----------------|
|---------------|----------------|

|         |                           |
|---------|---------------------------|
| getName | - Obtain a thread's name. |
|---------|---------------------------|

|             |                               |
|-------------|-------------------------------|
| getPriority | - Obtain a thread's priority. |
|-------------|-------------------------------|

|         |   |
|---------|---|
| isAlive | - Determine if a thread is still running. |
|---------|---|

|      |                                   |
|------|-----------------------------------|
| join | - Wait for a thread to terminate. |
|------|-----------------------------------|

|     |                               |
|-----|-------------------------------|
| run | - Entry point for the thread. |
|-----|-------------------------------|

|       |  |
|-------|--|
| sleep | - Suspend a thread for a period of time. |
|-------|--|

|       |   |
|-------|---|
| start | - Start a thread by calling its run method. |
|-------|---|

Thus far, all the examples in this book have used a single thread of execution. The remainder of this chapter explains how to use Thread and Runnable to create and manage threads, beginning with the one thread that all Java programs have: the main thread.

### **3.22.3. Creating a Thread**

In the most general sense, you create a thread by instantiating an object of type Thread. Java defines two ways in which this can be accomplished:

- You can implement the Runnable interface.
- You can extend the Thread class, itself.

The following two sections look at each method, in turn.

### **Implementing Runnable**

The easiest way to create a thread is to create a class that implements the Runnable interface. Runnable abstracts a unit of executable code. You can construct a thread on any object that implements Runnable. To implement Runnable, a class need only implement a single method called run( ), which is declared like this:

***public void run( )***

Inside run( ), you will define the code that constitutes the new thread. It is important to understand that run( ) can call other methods, use other classes, and declare variables, just like the main thread can. The only difference is that run( ) establishes the entry point for another, concurrent thread of execution within your program. This thread will end when run( ) returns.

After you create a class that implements Runnable, you will instantiate an object of type Thread from within that class. Thread defines several constructors. The one that we will use is shown here:

***Thread(Runnable threadOb, String threadName)***

In this constructor, threadOb is an instance of a class that implements the Runnable interface. This defines where execution of the thread will begin. The name of the new thread is specified by threadName.

After the new thread is created, it will not start running until you call its start( ) method, which is declared within Thread. In essence, start( ) executes a call to run( ). The start( ) method is shown here:

***void start( )***

Here is an example that creates a new thread and starts it running:

```
// Create a second thread.
class NewThread implements Runnable {
    Thread t;
    NewThread() {
        // Create a new, second thread
        t = new Thread(this, "Demo Thread");
        System.out.println("Child thread: " + t);
        t.start(); // Start the thread
    }
}
```



```

}
// This is the entry point for the second thread.
public void run() {
    try {
        for(int i = 5; i > 0; i—) {
            System.out.println("Child Thread: " + i);
            Thread.sleep(500);
        }
    } catch (InterruptedException e) {
        System.out.println("Child interrupted.");
    }
    System.out.println("Exiting child thread.");
}

class ThreadDemo {
    public static void main(String args[]) {
        new NewThread(); // create a new thread
        try {
            for(int i = 5; i > 0; i—) {
                System.out.println("Main Thread: " + i);
                Thread.sleep(1000);
            }
        } catch (InterruptedException e) {
            System.out.println("Main thread interrupted.");
        }
        System.out.println("Main thread exiting.");
    }
}

```

Inside NewThread's constructor, a new Thread object is created by the following statement:

```
t = new Thread(this, "Demo Thread");
```

Passing this as the first argument indicates that you want the new thread to call the run( ) method on this object. Next, start( ) is called, which starts the thread of execution beginning at the run( ) method. This causes the child thread's for loop to begin. After calling start( ), NewThread's constructor returns to main( ). When the main thread resumes, it enters its for loop. Both threads continue running, sharing the CPU, until their loops finish.

The output produced by this program is as follows:

*Child thread: Thread[Demo Thread,5,main]*

*Main Thread: 5*

*Child Thread: 5*

*Child Thread: 4*

*Main Thread: 4*

*Child Thread: 3*

*Child Thread: 2*

*Main Thread: 3*

*Child Thread: 1*

*Exiting child thread.*

*Main Thread: 2*

*Main Thread: 1*

*Main thread exiting.*

As mentioned earlier, in a multithreaded program, the main thread must be the last thread to finish running. If the main thread finishes before a child thread has completed, then the Java run-time system may "hang." The preceding program ensures that the main thread finishes last, because the main thread sleeps for 1,000 milliseconds between iterations, but the child thread sleeps for only 500 milliseconds. This causes the child thread to terminate earlier than the main thread. Shortly, you will see a better way to ensure that the main thread finishes last.

## **Extending Thread**

The second way to create a thread is to create a new class that extends Thread, and then to create an instance of that class. The extending class must override the run( ) method, which is the entry point for the new thread. It must also call start( ) to begin execution of the new thread. Here is the preceding program rewritten to extend Thread:

```
// Create a second thread by extending Thread  
class NewThread extends Thread {  
    NewThread() {  
        // Create a new, second thread  
        super("Demo Thread");  
        System.out.println("Child thread: " + this);  
        start(); // Start the thread  
    }  
}  
  
// This is the entry point for the second thread.  
public void run() {
```

```

try {
for(int i = 5; i > 0; i--) {
    System.out.println("Child Thread: " + i);
    Thread.sleep(500);
}
} catch (InterruptedException e) {
    System.out.println("Child interrupted.");
}
System.out.println("Exiting child thread.");
}

class ExtendThread {
    public static void main(String args[]) {
        new NewThread(); // create a new thread
        try {
            for(int i = 5; i > 0; i--) {
                System.out.println("Main Thread: " + i);
                Thread.sleep(1000);
            }
        } catch (InterruptedException e) {
            System.out.println("Main thread interrupted.");
        }
        System.out.println("Main thread exiting.");
    }
}

```

This program generates the same output as the preceding version. As you can see, the child thread is created by instantiating an object of `NewThread`, which is derived from `Thread`. Notice the call to `super( )` inside `NewThread`. This invokes the following form of the `Thread` constructor:

```
public Thread(String threadName)
```

Here, `threadName` specifies the name of the thread.

### 3.22.4. Thread Priorities

Thread priorities are used by the thread scheduler to decide when each thread should be allowed to run. In theory, higher-priority threads get more CPU time than lower-priority threads. In practice, the amount of CPU time that a thread gets often depends on several factors besides its priority. (For example, how an operating system implements multitasking can affect the relative availability of CPU time.) A higher-priority thread can also preempt a lower-priority one. For instance, when a

lower-priority thread is running and a higher-priority thread resumes (from sleeping or waiting on I/O, for example), it will preempt the lower-priority thread.

In theory, threads of equal priority should get equal access to the CPU. But you need to be careful. Remember, Java is designed to work in a wide range of environments. Some of those environments implement multitasking fundamentally differently than others. For safety, threads that share the same priority should yield control once in a while. This ensures that all threads have a chance to run under a nonpreemptive operating system.

In practice, even in nonpreemptive environments, most threads still get a chance to run, because most threads inevitably encounter some blocking situation, such as waiting for I/O. When this happens, the blocked thread is suspended and other threads can run. But, if you want smooth multithreaded execution, you are better off not relying on this. Also, some types of tasks are CPU-intensive. Such threads dominate the CPU. For these types of threads, you want to yield control occasionally, so that other threads can run. To set a thread's priority, use the `setPriority( )` method, which is a member of `Thread`. This is its general form:

```
final void setPriority(int level)
```

Here, `level` specifies the new priority setting for the calling thread. The value of `level` must be within the range `MIN_PRIORITY` and `MAX_PRIORITY`. Currently, these values are 1 and 10, respectively. To return a thread to default priority, specify `NORM_PRIORITY`, which is currently 5. These priorities are defined as final variables within `Thread`. You can obtain the current priority setting by calling the `getPriority( )` method of `Thread`, shown here:

```
final int getPriority( )
```

Implementations of Java may have radically different behavior when it comes to scheduling. The Windows 95/98/NT version works, more or less, as you would expect. However, other versions may work quite differently. Most of the inconsistencies arise when you have threads that are relying on preemptive behavior, instead of cooperatively giving up CPU time. The safest way to obtain predictable, cross-platform behavior with Java is to use threads that voluntarily give up control of the CPU.

### **3.22.5. Synchronization**

When two or more threads need access to a shared resource, they need some way to ensure that the resource will be used by only one thread at a time. The process by which this is achieved is called synchronization. As you will see, Java provides unique, language-level support for it. Because multithreading introduces an asynchronous behavior to your programs, there must be a way for you to enforce synchronicity when you need it. Key to synchronization is the concept of the monitor (also called a semaphore). A monitor is an object that is used as a mutually exclusive lock, or mutex. Only one thread can own a monitor at a given time. When a thread acquires a lock, it is said

to have entered the monitor. All other threads attempting to enter the locked monitor will be suspended until the first thread exits the monitor. These other threads are said to be waiting for the monitor. A thread that owns a monitor can reenter the same monitor if it so desires. You can synchronize your code in either of two ways. Both involve the use of the synchronized keyword, and both are examined here.

### **3.22.6. Interthread Communication**

The preceding examples unconditionally blocked other threads from asynchronous access to certain methods. This use of the implicit monitors in Java objects is powerful, but you can achieve a more subtle level of control through interprocess communication.

As you will see, this is especially easy in Java. As discussed earlier, multithreading replaces event loop programming by dividing your tasks into discrete and logical units. Threads also provide a secondary benefit: they do away with polling. Polling is usually implemented by a loop that is used to check some condition repeatedly. Once the condition is true, appropriate action is taken. This wastes CPU time. For example, consider the classic queuing problem, where one thread is producing some data and another is consuming it. To make the problem more interesting, suppose that the producer has to wait until the consumer is finished before it generates more data. In a polling system, the consumer would waste many CPU cycles while it waited for the producer to produce. Once the producer was finished, it would start polling, wasting more CPU cycles waiting for the consumer to finish, and so on. Clearly, this situation is undesirable.

To avoid polling, Java includes an elegant interprocess communication mechanism via the `wait()`, `notify()`, and `notifyAll()` methods. These methods are implemented as final methods in `Object`, so all classes have them. All three methods can be called only from within a synchronized method. Although conceptually advanced from a computer science perspective, the rules for using these methods are actually quite simple:

- `wait()` tells the calling thread to give up the monitor and go to sleep until some other thread enters the same monitor and calls `notify()`.
- `notify()` wakes up the first thread that called `wait()` on the same object.
- `notifyAll()` wakes up all the threads that called `wait()` on the same object. The highest priority thread will run first.

These methods are declared within `Object`, as shown here:

- `final void wait()` throws `InterruptedException`
- `final void notify()`
- `final void notifyAll()`

Additional forms of `wait()` exist that allow you to specify a period of time to wait. The following sample program incorrectly implements a simple form of the producer/consumer problem. It

consists of four classes: Q, the queue that you're trying to synchronize; Producer, the threaded object that is producing queue entries; Consumer, the threaded object that is consuming queue entries; and PC, the tiny class that creates the single Q, Producer, and Consumer.

// An incorrect implementation of a producer and consumer.

```
class Q {
    int n;

    synchronized int get() {
        System.out.println("Got: " + n);
        return n;
    }
}

synchronized void put(int n) {
    this.n = n;
    System.out.println("Put: " + n);
}

class Producer implements Runnable {
    Q q;
    Producer(Q q) {
        - 205 -
    }
    this.q = q;
    new Thread(this, "Producer").start();
    public void run() {
        int i = 0;
    }
    }
    while(true) {
        q.put(i++);
    }
}

class Consumer implements Runnable {
    Q q;
    Consumer(Q q) {
        this.q = q;
        new Thread(this, "Consumer").start();
    }
}
```

```

    }
    public void run() {
        while(true) {
            q.get();
        }
    }
}
class PC {
    public static void main(String args[]) {
        Q q = new Q();
        new Producer(q);
        new Consumer(q);
        System.out.println("Press Control-C to stop.");
    }
}

```

Although the put( ) and get( ) methods on Q are synchronized, nothing stops the producer from overrunning the consumer, nor will anything stop the consumer from consuming the same queue value twice. Thus, you get the erroneous output shown here (the exact output will vary with processor speed and task load):

```

Put:1
Got:1
Got:1
Got:1
Got:1
Got:1
Put:2
Put:3
Put:4
Put:5
Put:6
Put:7
Got:7

```

As you can see, after the producer put 1, the consumer started and got the same 1 five times in a row. Then, the producer resumed and produced 2 through 7 without letting the consumer have a chance to consume them. The proper way to write this program in Java is to use wait( ) and notify( ) to signal in both directions, as shown here:

// A correct implementation of a producer and consumer.

```
class Q {
    int n;
    boolean valueSet = false;
    synchronized int get() {
        if(!valueSet)
            try {
                wait();
            } catch(InterruptedException e) {
                System.out.println("InterruptedException caught");
            }
        System.out.println("Got: " + n);
        valueSet = false;
        notify();
    }
    return n;
    synchronized void put(int n) {
        if(valueSet)
            try {
                wait();
            } catch(InterruptedException e) {
                System.out.println("InterruptedException caught");
            }
    }
    }
    this.n = n;
    valueSet = true;
    System.out.println("Put: " + n);
    notify();
    class Producer implements Runnable {
        Q q;
        Producer(Q q) {
            this.q = q;
            new Thread(this, "Producer").start();
        }
    }
}
```



```

public void run() {
    int i = 0;
}
}
while(true) {
    q.put(i++);
}
class Consumer implements Runnable {
    Q q;
    Consumer(Q q) {
        this.q = q;
        new Thread(this, "Consumer").start();
    }
}
public void run() {
    while(true) {
        q.get();
    }
}
class PCFixed {
    public static void main(String args[]) {
        Q q = new Q();
        new Producer(q);
        new Consumer(q);
        System.out.println("Press Control-C to stop.");
    }
}

```

Inside `get()`, `wait()` is called. This causes its execution to suspend until the Producer notifies you that some data is ready. When this happens, execution inside `get()` resumes. After the data has been obtained, `get()` calls `notify()`. This tells Producer that it is okay to put more data in the queue. Inside `put()`, `wait()` suspends execution until the Consumer has removed the item from the queue. When execution resumes, the next item of data is put in the queue, and `notify()` is called. This tells the Consumer that it should now remove it.

Here is some output from this program, which shows the clean synchronous behavior:

Put:1

Got:1

Put:2

Got:2

Put:3

Got:3

Put:4

Got:4

Put:5

Got:5

### **3.22.7. Deadlock**

A special type of error that you need to avoid that relates specifically to multitasking is deadlock, which occurs when two threads have a circular dependency on a pair of synchronized objects. For example, suppose one thread enters the monitor on object X and another thread enters the monitor on object Y. If the thread in X tries to call any synchronized method on Y, it will block as expected. However, if the thread in Y, in turn, tries to call any synchronized method on X, the thread waits forever, because to access X, it would have to release its own lock on Y so that the first thread could complete.

Deadlock is a difficult error to debug for two reasons:

- In general, it occurs only rarely, when the two threads time-slice in just the right way.
- It may involve more than two threads and two synchronized objects. (That is, deadlock can occur through a more convoluted sequence of events than just described.)

### **3.22.8. Suspending, Resuming, and Stopping Threads**

Sometimes, suspending execution of a thread is useful. For example, a separate thread can be used to display the time of day. If the user doesn't want a clock, then its thread can be suspended. Whatever the case, suspending a thread is a simple matter. Once suspended, restarting the thread is also a simple matter.

## **Module IV**

### **Files and I/O Streams**

The `io` package supports Java's basic I/O (input/output) system, including file I/O.

#### **4.1. I/O Basics**

As you may have noticed while reading the preceding sections, not much use has been made of I/O in the example programs. In fact, aside from `print( )` and `println( )`, none of the I/O methods have been used significantly. The reason is simple: most real applications of Java are not text-based, console programs. Rather, they are graphically oriented applets that rely upon Java's Abstract Window Toolkit (AWT) for interaction with the user. Although text-based programs are excellent as teaching examples, they do not constitute an important use for Java in the real world. Also, Java's support for console I/O is limited and somewhat awkward to use—even in simple example programs. Text-based console I/O is just not very important to Java programming.

The preceding paragraph notwithstanding, Java does provide strong, flexible support for I/O as it relates to files and networks. Java's I/O system is cohesive and consistent. In fact, once you understand its fundamentals, the rest of the I/O system is easy to master.

#### **4.2. Streams**

Java programs perform I/O through streams. A stream is an abstraction that either produces or consumes information. A stream is linked to a physical device by the Java I/O system. All streams behave in the same manner, even if the actual physical devices to which they are linked differ. Thus, the same I/O classes and methods can be applied to any type of device. This means that an input stream can abstract many different kinds of input: from a disk file, a keyboard, or a network socket. Likewise, an output stream may refer to the console, a disk file, or a network connection. Streams are a clean way to deal with input/output without having every part of your code understand the difference between a keyboard and a network, for example. Java implements streams within class hierarchies defined in the `java.io` package.

##### **4.2.1. Byte Streams and Character Streams**

Java 2 defines two types of streams: byte and character. Byte streams provide a convenient means for handling input and output of bytes. Byte streams are used, for example, when reading or writing binary data. Character streams provide a convenient means for handling input and output of characters. They use Unicode and, therefore, can be internationalized. Also, in some cases, character streams are more efficient than byte streams.

The original version of Java (Java 1.0) did not include character streams and, thus, all I/O was byte-oriented. Character streams were added by Java 1.1, and certain byte-oriented classes and methods were deprecated. This is why older code that doesn't use character streams should be updated to take advantage of them, where appropriate.

One other point: at the lowest level, all I/O is still byte-oriented. The character-based streams simply provide a convenient and efficient means for handling characters. An overview of both byte-oriented streams and character-oriented streams is presented in the following sections.

### **The Byte Stream Classes**

Byte streams are defined by using two class hierarchies. At the top are two abstract classes: `InputStream` and `OutputStream`. Each of these abstract classes has several concrete subclasses, that handle the differences between various devices, such as disk files, network connections, and even memory buffers. The byte stream classes are shown below. Remember, to use the stream classes, you must import `java.io`.

- `BufferedInputStream` Buffered input stream
- `BufferedOutputStream` Buffered output stream
- `ByteArrayInputStream` Input stream that reads from a byte array
- `ByteArrayOutputStream` Output stream that writes to a byte array
- `DataInputStream` An input stream that contains methods for reading the Java standard data types
- `DataOutputStream` An output stream that contains methods for writing the Java standard data types
- `FileInputStream` Input stream that reads from a file
- `FileOutputStream` Output stream that writes to a file
- `FilterInputStream` Implements `InputStream`
- `FilterOutputStream` Implements `OutputStream`
- `InputStream` Abstract class that describes stream input
- `OutputStream` Abstract class that describes stream output
- `PipedInputStream` Input pipe
- `PipedOutputStream` Output pipe
- `PrintStream` Output stream that contains `print( )` and `println( )`
- `PushbackInputStream` Input stream that supports one-byte "unget," which returns a byte to the input stream
- `RandomAccessFile` Supports random access file I/O
- `SequenceInputStream` Input stream that is a combination of two or more input streams that will be read sequentially, one after the other

### **The Character Stream Classes**

Character streams are defined by using two class hierarchies. At the top are two abstract classes, `Reader` and `Writer`. These abstract classes handle Unicode character streams.

Java has several concrete subclasses of each of these. The character stream classes are shown below. The abstract classes `Reader` and `Writer` define several key methods that the other stream classes implement. Two of the most important methods are `read( )` and `write( )`, which read and write characters of data, respectively. These methods are overridden by derived stream classes.

- `BufferedReader` Buffered input character stream
- `BufferedWriter` Buffered output character stream
- `CharArrayReader` Input stream that reads from a character array
- `CharArrayWriter` Output stream that writes to a character array
- `FileReader` Input stream that reads from a file
- `FileWriter` Output stream that writes to a file
- `FilterReader` Filtered reader
- `FilterWriter` Filtered writer
- `InputStreamReader` Input stream that translates bytes to characters
- `LineNumberReader` Input stream that counts lines
- `OutputStreamWriter` Output stream that translates characters to bytes
- `PipedReader` Input pipe
- `PipedWriter` Output pipe
- `PrintWriter` Output stream that contains `print( )` and `println( )`
- `PushbackReader` Input stream that allows characters to be returned to the input stream
- `Reader` Abstract class that describes character stream input
- `StringReader` Input stream that reads from a string
- `StringWriter` Output stream that writes to a string
- `Writer` Abstract class that describes character stream output

### **Reading Console Input**

In Java, console input is accomplished by reading from `System.in`. To obtain a character-based stream that is attached to the console, you wrap `System.in` in a `BufferedReader` object, to create a character stream. `BufferedReader` supports a buffered input stream. Its most commonly used constructor is shown here:

*`BufferedReader(Reader inputReader)`*

Here, `inputReader` is the stream that is linked to the instance of `BufferedReader` that is being created. `Reader` is an abstract class. One of its concrete subclasses is `InputStreamReader`, which converts bytes to characters. To obtain an `InputStreamReader` object that is linked to `System.in`, use the following constructor:

*`InputStreamReader(InputStream inputStream)`*

Because `System.in` refers to an object of type `InputStream`, it can be used for `InputStream`. Putting it all together, the following line of code creates a `BufferedReader` that is connected to the keyboard:

```
BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
```

After this statement executes, `br` is a character-based stream that is linked to the console through `System.in`.

## Reading Characters

To read a character from a `BufferedReader`, use `read( )`. The version of `read( )` that we will be using is

```
int read( ) throws IOException
```

Each time that `read( )` is called, it reads a character from the input stream and returns it as an integer value. It returns `-1` when the end of the stream is encountered. As you can see, it can throw an `IOException`. The following program demonstrates `read( )` by reading characters from the console until the user types a "q":

```
// Use a BufferedReader to read characters from the console.  
import java.io.*;  
class BRRead {  
public static void main(String args[]  
throws IOException  
{  
char c;  
BufferedReader br = new  
BufferedReader(new InputStreamReader(System.in));  
System.out.println("Enter characters, 'q' to quit.");  
// read characters  
do {  
c = (char) br.read();  
System.out.println(c);  
} while(c != 'q');  
}  
}
```

Here is a sample run:

Enter characters, 'q' to quit.

123abcq

1

2

3  
a  
b  
c  
q

This output may look a little different from what you expected, because `System.in` is line buffered, by default. This means that no input is actually passed to the program until you press ENTER. As you can guess, this does not make `read( )` particularly valuable for interactive, console input.

## Reading Strings

To read a string from the keyboard, use the version of `readLine( )` that is a member of the `BufferedReader` class. Its general form is shown here:

*String readLine( ) throws IOException*

As you can see, it returns a `String` object. The following program demonstrates `BufferedReader` and the `readLine( )` method; the program reads and displays lines of text until you enter the word "stop":

```
// Read a string from console using a BufferedReader.
import java.io.*;

class BRReadLines {
    public static void main(String args[]) throws IOException
    {
        // create a BufferedReader using System.in
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        String str;
        System.out.println("Enter lines of text.");
        System.out.println("Enter 'stop' to quit.");
        do {
            str = br.readLine();
            System.out.println(str);
        } while(!str.equals("stop"));
    }
}
```

## Writing Console Output

Console output is most easily accomplished with `print( )` and `println( )`, described earlier, which are used in most of the examples in this book. These methods are defined by the class `PrintStream` (which is the type of the object referenced by `System.out`). Even though `System.out` is a byte stream, using it for simple program output is still acceptable.

Because `PrintStream` is an output stream derived from `OutputStream`, it also implements the low-level method `write( )`. Thus, `write( )` can be used to write to the console. The simplest form of `write()` defined by `PrintStream` is shown here:

*`void write(int byteval) throws IOException`*

This method writes to the file the byte specified by `byteval`. Although `byteval` is declared as an integer, only the low-order eight bits are written. Here is a short example that uses `write( )` to output the character "A" followed by a newline to the screen:

```
// Demonstrate System.out.write().  
class WriteDemo {  
    public static void main(String args[]) {  
        int b;  
        b = 'A';  
        System.out.write(b);  
        System.out.write("\n");  
    }  
}
```

You will not often use `write( )` to perform console output (although doing so might be useful in some situations), because `print( )` and `println( )` are substantially easier to use.

### **4.3. Reading and Writing Files**

Java provides a number of classes and methods that allow you to read and write files. In Java, all files are byte-oriented, and Java provides methods to read and write bytes from and to a file. However, Java allows you to wrap a byte-oriented file stream within a character-based object. This technique is described in Part II. This chapter examines the basics of file I/O.

Two of the most often-used stream classes are `FileInputStream` and `FileOutputStream`, which create byte streams linked to files. To open a file, you simply create an object of one of these classes, specifying the name of the file as an argument to the constructor. While both classes support additional, overridden constructors, the following are the forms that we will be using:

1. `FileInputStream(String fileName)` throws `FileNotFoundException`
2. `FileOutputStream(String fileName)` throws `FileNotFoundException`

Here, `fileName` specifies the name of the file that you want to open. When you create an input stream, if the file does not exist, then `FileNotFoundException` is thrown. For output streams, if the file cannot be created, then `FileNotFoundException` is thrown. When an output file is opened, any preexisting file by the same name is destroyed. Note In earlier versions of Java, `FileOutputStream( )` threw an `IOException` when an output file could not be created.



When you are done with a file, you should close it by calling `close( )`. It is defined by both `FileInputStream` and `FileOutputStream`, as shown here:

*`void close( ) throws IOException`*

To read from a file, you can use a version of `read( )` that is defined within `FileInputStream`. The one that we will use is shown here:

*`int read( ) throws IOException`*

Each time that it is called, it reads a single byte from the file and returns the byte as an integer value. `read( )` returns `-1` when the end of the file is encountered. It can throw an `IOException`.

The following program uses `read( )` to input and display the contents of a text file, the name of which is specified as a command-line argument. Note the try/catch blocks that handle the two errors that might occur when this program is used—the specified file not being found or the user forgetting to include the name of the file. You can use this same approach whenever you use command-line arguments.

```
/* Display a text file.  
To use this program, specify the name  
of the file that you want to see.  
For example, to see a file called TEST.TXT,  
use the following command line.  
  
java ShowFile TEST.TXT  
  
*/  
  
import java.io.*;  
class ShowFile {  
public static void main(String args[])  
throws IOException  
{  
int i;  
FileInputStream fin;  
try {  
fin = new FileInputStream(args[0]);  
} catch(FileNotFoundException e) {  
System.out.println("File Not Found");  
return;  
} catch(ArrayIndexOutOfBoundsException e) {  
System.out.println("Usage: ShowFile File");  
return;
```

```

}
// read characters until EOF is encountered
do {
    i = fin.read();
    if(i != -1) System.out.print((char) i);
} while(i != -1);
fin.close();
}
}

```

**To write to a file**, you will use the write( ) method defined by FileOutputStream. Its simplest form is shown here:

*void write(int byteval) throws IOException*

This method writes the byte specified by byteval to the file. Although byteval is declared as an integer, only the low-order eight bits are written to the file. If an error occurs during writing, an IOException is thrown. The next example uses write( ) to copy a text file:

```

/* Copy a text file.
To use this program, specify the name
of the source file and the destination file.
For example, to copy a file called FIRST.TXT
to a file called SECOND.TXT, use the following
command line.
*/

java CopyFile FIRST.TXT SECOND.TXT
import java.io.*;
class CopyFile {
    public static void main(String args[])
        throws IOException
    {
        int i;
        FileInputStream fin;
        FileOutputStream fout;
        try {
            // open input file
            try {
                fin = new FileInputStream(args[0]);

```

```

    } catch(FileNotFoundException e) {
        System.out.println("Input File Not Found");
        return;
    }
    // open output file
    try {
        fout = new FileOutputStream(args[1]);
    } catch(FileNotFoundException e) {
        System.out.println("Error Opening Output File");
        return;
    }
    } catch(ArrayIndexOutOfBoundsException e) {
        System.out.println("Usage: CopyFile From To");
        return;
    }
    // Copy File
    try {
        do {
            i = fin.read();
            if(i != -1) fout.write(i);
        } while(i != -1);
    } catch(IOException e) {
        System.out.println("File Error");
    }
    fin.close();
    fout.close();
}
}

```

Notice the way that potential I/O errors are handled in this program and in the preceding ShowFile program. Unlike most other computer languages, including C and C++, which use error codes to report file errors, Java uses its exception handling mechanism. Not only does this make file handling cleaner, but it also enables Java to easily differentiate the end- of-file condition from file errors when input is being performed. In C/C++, many input functions return the same value when an error occurs and when the end of the file is reached. (That is, in C/C++, an EOF condition often is mapped to the same value as an input error.) This usually means that the programmer must

include extra program statements to determine which event actually occurred. In Java, errors are passed to your program via exceptions, not by values returned by `read()`. Thus, when `read()` returns `-1`, it means only one thing: the end of the file has been encountered.

#### 4.4. Filter Streams

As mentioned earlier, The `InputStream` and `OutputStream` classes are used for reading from and writing to byte streams, respectively. In this section, you are going to learn of data transforming and manipulating using Filter Streams.

Like I/O streams, Filter streams are also used to manipulate data reading from an underlying stream. Apart from this, it allows the user to make a chain using multiple input stream so that, the operations that are to be applied on this chain, may create a combine effects on several filters. By using these streams, there is no need to convert the data from byte to char while writing to a file. These are the more powerful streams than the other streams of Java.

There are two streams that are derived from the I/O stream class:

- `FilterInputStream`
- `FilterOutputStream`

#### 4.5 Random Access File

The `RandomAccessFile` class in the Java IO API allows you to move around a file and read from it or write to it as you please. You can replace existing parts of a file too. This is not possible with the `FileInputStream` or `FileOutputStream`.

##### Creating a RandomAccessFile

Before you can work with the `RandomAccessFile` class you must instantiate it. Here is how that looks:

```
RandomAccessFile file = new RandomAccessFile("c:\\data\\file.txt", "rw");
```

Notice the second input parameter to the constructor: `"rw"`. This is the mode you want to open file in. `"rw"` means read/write mode. Check the JavaDoc for more details about what modes you can open a `RandomAccessFile` in.

##### Moving Around a RandomAccessFile

To read or write at a specific location in a `RandomAccessFile` you must first position the file pointer at the location to read or write. This is done using the `seek()` method. The current position of the file pointer can be obtained by calling the `getFilePointer()` method.

Here is a simple example:

```
RandomAccessFile file = new RandomAccessFile("c:\\data\\file.txt", "rw");  
file.seek(200);  
long pointer = file.getFilePointer();  
file.close();
```

### Reading from a RandomAccessFile

Reading from a RandomAccessFile is done using one of its many read() methods. Here is a simple example:

```
RandomAccessFile file = new RandomAccessFile("c:\\data\\file.txt", "rw");  
int aByte = file.read();  
file.close();
```

The read() method reads the byte located at the position in the file currently pointed to by the file pointer in the RandomAccessFile instance.

Here is a thing the JavaDoc forgets to mention: The read() method increments the file pointer to point to the next byte in the file after the byte just read! This means that you can continue to call read() without having to manually move the file pointer.

### Writing to a RandomAccessFile

Writing to a RandomAccessFile can be done using one of its many write() methods. Here is a simple example:

```
RandomAccessFile file = new RandomAccessFile("c:\\data\\file.txt", "rw");  
file.write("Hello World".getBytes());  
file.close();
```

Just like with the read() method the write() method advances the file pointer after being called. That way you don't have to constantly move the file pointer to write data to a new location in the file.

## 4.6 Serialization

Serialization is the process of saving an object in a storage medium (such as a file, or a memory buffer) or to transmit it over a network connection in binary form. The serialized objects are JVM independent and can be re-serialized by any JVM. In this case the "in memory" java objects state are converted into a byte stream. This type of the file can not be understood by the user. It is a special type of object i.e. reused by the JVM (Java Virtual Machine). This process of serializing an object is also called deflating or marshalling an object.

The given example shows the implementation of serialization to an object. This program takes a file name that is machine understandable and then serializes the object. The object to be serialized must implement java.io.Serializable class. Default serialization mechanism for an object writes the class of the object, the class signature, and the values of all non-transient and non-static fields.

class ObjectOutputStream extends java.io.OutputStream implements ObjectOutput, ObjectOutput interface extends the DataOutput interface and adds methods for serializing objects and writing bytes to the file. The ObjectOutputStream extends java.io.OutputStream and implements ObjectOutput interface. It serializes objects, arrays, and other values to a stream. Thus the constructor of ObjectOutputStream is written as:

```
ObjectOutput ObjOut = new ObjectOutputStream(new FileOutputStream(f));
```

Above code has been used to create the instance of the `ObjectOutput` class with the `ObjectOutputStream()` constructor which takes the instance of the `FileOutputStream` as a parameter. The `ObjectOutput` interface is used by implementing the `ObjectOutputStream` class. The `ObjectOutputStream` is constructed to serialize the object.

#### **4.7. Applet Fundamentals**

All of the preceding examples in this book have been Java applications. However, applications constitute only one class of Java programs. The other type of program is the applet. As mentioned in earlier, applets are small applications that are accessed on an Internet server, transported over the Internet, automatically installed, and run as part of a Web document. After an applet arrives on the client, it has limited access to resources, so that it can produce an arbitrary multimedia user interface and run complex computations without introducing the risk of viruses or breaching data integrity. However, the fundamentals connected to the creation of an applet are presented here, because applets are not structured in the same way as the programs that have been used thus far. As you will see, applets differ from applications in several key areas.

Let's begin with the simple applet shown here:

```
import java.awt.*;
import java.applet.*;
public class SimpleApplet extends Applet {
    public void paint(Graphics g) {
        g.drawString("A Simple Applet", 20, 20);
    }
}
```

This applet begins with two import statements. The first imports the Abstract Window Toolkit (AWT) classes. Applets interact with the user through the AWT, not through the console-based I/O classes. The AWT contains support for a window-based, graphical interface. As you might expect, the AWT is quite large and sophisticated, and a complete discussion of it consumes several chapters in Part II of this book. Fortunately, this simple applet makes very limited use of the AWT. The second import statement imports the applet package, which contains the class `Applet`. Every applet that you create must be a subclass of `Applet`.

The next line in the program declares the class `SimpleApplet`. This class must be declared as `public`, because it will be accessed by code that is outside the program. Inside `SimpleApplet`, `paint()` is declared. This method is defined by the AWT and must be overridden by the applet. `paint()` is called each time that the applet must redisplay its output. Whatever the cause, whenever the applet must redraw its output, `paint()` is called. The `paint()` method has one parameter of type `Graphics`.

This parameter contains the graphics context, which describes the graphics environment in which the applet is running. This context is used whenever output to the applet is required.

Inside `paint( )` is a call to `drawString( )`, which is a member of the `Graphics` class. This method outputs a string beginning at the specified X,Y location. It has the following general form:

```
void drawString(String message, int x, int y)
```

Here, `message` is the string to be output beginning at `x,y`. In a Java window, the upper-left corner is location 0,0. The call to `drawString( )` in the applet causes the message "A Simple Applet" to be displayed beginning at location 20,20.

Notice that the applet does not have a `main( )` method. Unlike Java programs, applets do not begin execution at `main( )`. In fact, most applets don't even have a `main( )` method. Instead, an applet begins execution when the name of its class is passed to an applet viewer or to a network browser.

After you enter the source code for `SimpleApplet`, compile in the same way that you have been compiling programs. However, running `SimpleApplet` involves a different process. In fact, there are two ways in which you can run an applet:

- Executing the applet within a Java-compatible Web browser, such as Netscape Navigator.
- Using an applet viewer, such as the standard JDK tool, `appletviewer`. An applet viewer executes your applet in a window. This is generally the fastest and easiest way to test your applet.

Each of these methods is described next.

To execute an applet in a Web browser, you need to write a short HTML text file that contains the appropriate `APPLET` tag. Here is the HTML file that executes

`SimpleApplet`:

```
<applet code="SimpleApplet" width=200 height=60>  
</applet>
```

The width and height statements specify the dimensions of the display area used by the applet. (The `APPLET` tag contains several other options that are examined more closely in Part II.) After you create this file, you can execute your browser and then load this file, which causes `SimpleApplet` to be executed.

To execute `SimpleApplet` with an applet viewer, you may also execute the HTML file shown earlier. For example, if the preceding HTML file is called `RunApp.html`, then the following command line will run `SimpleApplet`:

```
C:\>appletviewer RunApp.html
```

However, a more convenient method exists that you can use to speed up testing. Simply include a comment at the head of your Java source code file that contains the `APPLET` tag. By doing so, your code is documented with a prototype of the necessary HTML statements, and you can test your

compiled applet merely by starting the applet viewer with your Java source code file. If you use this method, the SimpleApplet source file looks like this:

```
import java.awt.*;
import java.applet.*;
/*
<applet code="SimpleApplet" width=200 height=60>
</applet>
*/
public class SimpleApplet extends Applet {
    public void paint(Graphics g) {
        g.drawString("A Simple Applet", 20, 20);
    }
}
```

In general, you can quickly iterate through applet development by using these three steps:

- Edit a Java source file.
- Compile your program.
- Execute the applet viewer, specifying the name of your applet's source file. The applet viewer will encounter the APPLET tag within the comment and execute your applet.

The window produced by SimpleApplet, as displayed by the applet viewer, is shown in the following illustration: While the subject of applets is more fully discussed later in this book, here are the key points that you should remember now:

1. Applets do not need a main( ) method.
2. Applets must be run under an applet viewer or a Java-compatible browser.
3. User I/O is not accomplished with Java's stream I/O classes. Instead, applets use the interface provided by the AWT.

#### **4.7.1. Applet class**

All applets are subclasses of Applet. Thus, all applets must import java.applet. Applets must also import java.awt. Recall that AWT stands for the Abstract Window Toolkit. Since all applets run in a window, it is necessary to include support for that window. Applets are not executed by the console-based Java run-time interpreter. Rather, they are executed by either a Web browser or an applet viewer. The figures shown in this chapter were created with the standard applet viewer, called appletviewer, provided by the JDK. But you can use any applet viewer or browser you like.

#### **4.7.2. An Applet Skeleton**

All but the most trivial applets override a set of methods that provides the basic mechanism by which the browser or applet viewer interfaces to the applet and controls its execution. Four of these



methods—`init()`, `start()`, `stop()`, and `destroy()`—are defined by `Applet`. Another, `paint()`, is defined by the `AWT Component` class. Default implementations for all of these methods are provided. Applets do not need to override those methods they do not use. However, only very simple applets will not need to define all of them. These five methods can be assembled into the skeleton shown here:

```
// An Applet skeleton.
import java.awt.*;
import java.applet.*;
/*
<applet code="AppletSkel" width=300 height=100>
</applet>
*/
public class AppletSkel extends Applet {
    // Called first.
    public void init() {
        // initialization
    }
    /* Called second, after init().
    the applet is restarted. */
    public void start() {
        // start or resume execution
    }
    Also called whenever
    // Called when the applet is stopped.
    public void stop() {
        // suspends execution
    }
    /* Called when applet is terminated.
    method executed. */
    public void destroy() {
        // perform shutdown activities
        This is the last
    }
    // Called when an applet's window must be restored.
    public void paint(Graphics g) {
```

```
// redisplay contents of window  
}  
}
```

Although this skeleton does not do anything, it can be compiled and run. When run, it generates the following window when viewed with an applet viewer:

#### **4.7.3. Applet Initialization and Termination**

It is important to understand the order in which the various methods shown in the skeleton are called. When an applet begins, the AWT calls the following methods, in this sequence: *init()*, *start()*, *paint()* When an applet is terminated, the following sequence of method calls takes place: *stop()*, *destroy()* Let's look more closely at these methods.

##### **init()**

The *init()* method is the first method to be called. This is where you should initialize variables. This method is called only once during the run time of your applet.

##### **start()**

The *start()* method is called after *init()*. It is also called to restart an applet after it has been stopped. Whereas *init()* is called once—the first time an applet is loaded—*start()* is called each time an applet's HTML document is displayed onscreen. So, if a user leaves a web page and comes back, the applet resumes execution at *start()*.

##### **paint()**

The *paint()* method is called each time your applet's output must be redrawn. This situation can occur for several reasons. For example, the window in which the applet is running may be overwritten by another window and then uncovered. Or the applet window may be minimized and then restored. *paint()* is also called when the applet begins execution. Whatever the cause, whenever the applet must redraw its output, *paint()* is called. The *paint()* method has one parameter of type *Graphics*. This parameter will contain the graphics context, which describes the graphics environment in which the applet is running. This context is used whenever output to the applet is required.

##### **stop()**

The *stop()* method is called when a web browser leaves the HTML document containing the applet—when it goes to another page, for example. When *stop()* is called, the applet is probably running. You should use *stop()* to suspend threads that don't need to run when the applet is not visible. You can restart them when *start()* is called if the user returns to the page.

##### **destroy()**

The `destroy()` method is called when the environment determines that your applet needs to be removed completely from memory. At this point, you should free up any resources the applet may be using. The `stop()` method is always called before `destroy()`.

#### **4.7.4.The HTML APPLET Tag**

The `APPLET` tag is used to start an applet from both an HTML document and from an applet viewer. An applet viewer will execute each `APPLET` tag that it finds in a separate window, while web browsers like Netscape Navigator, Internet Explorer, and HotJava will allow many applets on a single page. So far, we have been using only a simplified form of the `APPLET` tag. Now it is time to take a closer look at it.

The syntax for the standard `APPLET` tag is shown here. Bracketed items are optional.

```
< APPLET
[CODEBASE = codebaseURL]
CODE = appletFile
[ALT = alternateText]
[NAME = appletInstanceName]
WIDTH = pixels HEIGHT = pixels
[ALIGN = alignment]
[VSPACE = pixels] [HSPACE = pixels]
>
[< PARAM NAME = AttributeName VALUE = AttributeValue>]
[< PARAM NAME = AttributeName2 VALUE = AttributeValue>]
...
[HTML Displayed in the absence of Java]
</APPLET>
```

Let's take a look at each part now.

#### **CODEBASE**

`CODEBASE` is an optional attribute that specifies the base URL of the applet code, which is the directory that will be searched for the applet's executable class file (specified by the `CODE` tag). The HTML document's URL directory is used as the `CODEBASE` if this attribute is not specified. The `CODEBASE` does not have to be on the host from which the HTML document was read.

#### **CODE**

`CODE` is a required attribute that gives the name of the file containing your applet's compiled `.class` file. This file is relative to the code base URL of the applet, which is the directory that the HTML file was in or the directory indicated by `CODEBASE` if set.

#### **ALT**

The ALT tag is an optional attribute used to specify a short text message that should be displayed if the browser understands the APPLET tag but can't currently run Java applets. This is distinct from the alternate HTML you provide for browsers that don't support applets.

### **NAME**

NAME is an optional attribute used to specify a name for the applet instance. Applets must be named in order for other applets on the same page to find them by name and communicate with them. To obtain an applet by name, use `getApplet( )`, which is defined by the `AppletContext` interface.

### **WIDTH and HEIGHT**

WIDTH and HEIGHT are required attributes that give the size (in pixels) of the applet display area.

### **ALIGN**

ALIGN is an optional attribute that specifies the alignment of the applet. This attribute is treated the same as the HTML IMG tag with these possible values: LEFT, RIGHT, TOP, BOTTOM, MIDDLE, BASELINE, TEXTTOP, ABSMIDDLE, and ABSBOTTOM.

### **VSPACE and HSPACE**

These attributes are optional. VSPACE specifies the space, in pixels, above and below the applet. HSPACE specifies the space, in pixels, on each side of the applet. They're treated the same as the IMG tag's VSPACE and HSPACE attributes.

### **PARAM NAME and VALUE**

The PARAM tag allows you to specify applet-specific arguments in an HTML page. Applets access their attributes with the `getParameter( )` method.

## **Module V**

### **5.1. Introducing the AWT**

The AWT contains numerous classes and methods that allow you to create and manage windows. A full description of the AWT would easily fill an entire book. Therefore, it is not possible to describe in detail every method, instance variable, or class contained in the AWT.

Here , you will learn how to create and manage windows, manage fonts, output text, and utilize graphics.

Although the main purpose of the AWT is to support applet windows, it can also be used to create stand-alone windows that run in a GUI environment, such as Windows. Most of the examples are contained in applets, so to run them, you need to use an applet viewer or a Java-compatible Web browser. A few examples will demonstrate the creation of stand-alone, windowed programs.

#### **5.1.1. AWT Classes**

The AWT classes are contained in the java.awt package. It is one of Java's largest packages. Fortunately, because it is logically organized in a top-down, hierarchical fashion, it is easier to understand and use than you might at first believe. Table lists some of the AWT classes.

- AWTEvent Encapsulates AWT events.
- BorderLayout The border layout manager. Border layouts use five components: North, South, East, West, and Center.
- Button Creates a push button control.
- Checkbox Creates a check box control.
- Component An abstract superclass for various AWT components.
- Container A subclass of Component that can hold other components.
- Dialog Creates a dialog window.
- Event Encapsulates events.
- Font Encapsulates a type font.
- Frame Creates a standard window that has a title bar, resize corners, and a menu bar.
- Graphics Encapsulates the graphics context. This context is used by the various output methods to display output in a window.
- GridLayout The grid layout manager. Grid layout displays components in a two-dimensional grid.

- Image Encapsulates graphical images.
- Menu Creates a pull-down menu.
- Panel The simplest concrete subclass of Container.
- Scrollbar Creates a scroll bar control.
- TextArea Creates a multiline edit control.
- TextComponent A superclass for TextArea and TextField.
- TextField Creates a single-line edit control.
- Window Creates a window with no frame, no menu bar, and no title.

## **5.2. Window Fundamentals**

The AWT defines windows according to a class hierarchy that adds functionality and specificity with each level. The two most common windows are those derived from Panel, which is used by applets, and those derived from Frame, which creates a standard window. Much of the functionality of these windows is derived from their parent classes. Thus, a description of the class hierarchies relating to these two classes is fundamental to their understanding.

### **5.2.1. Component**

At the top of the AWT hierarchy is the Component class. Component is an abstract class that encapsulates all of the attributes of a visual component. All user interface elements that are displayed on the screen and that interact with the user are subclasses of Component. It defines over a hundred public methods that are responsible for managing events, such as mouse and keyboard input, positioning and sizing the window, and repainting.

### **5.2.2. Container**

The Container class is a subclass of Component. It has additional methods that allow other Component objects to be nested within it. Other Container objects can be stored inside of a Container (since they are themselves instances of Component). A container is responsible for laying out (that is, positioning) any components that it contains.

### **5.2.3. Panel**

The Panel class is a concrete subclass of Container. It doesn't add any new methods; it simply implements Container. A Panel may be thought of as a recursively nestable, concrete screen component. Panel is the superclass for Applet. When screen output is directed to an applet, it is drawn on the surface of a Panel object. In essence, a Panel is a window that does not contain a title bar, menu bar, or border. This is why you don't see these items when an applet is run inside

a browser. When you run an applet using an applet viewer, the applet viewer provides the title and border. Other components can be added to a Panel object by its add( ) method (inherited from Container).

#### **5.2.4. Window**

The Window class creates a top-level window. A top-level window is not contained within any other object; it sits directly on the desktop. Generally, you won't create Window objects directly. Instead, you will use a subclass of Window called Frame, described next.

#### **5.2.5. Frame**

Frame encapsulates what is commonly thought of as a "window." It is a subclass of Window and has a title bar, menu bar, borders, and resizing corners. If you create a Frame object from within an applet, it will contain a warning message, such as "Warning:

Applet Window," to the user that an applet window has been created. This message warns users that the window they see was started by an applet and not by software running on their computer. (An applet that could masquerade as a host-based application could be used to obtain passwords and other sensitive information without the user's knowledge.) When a Frame window is created by a program rather than an applet, a normal window is created.

#### **5.2.6. Canvas**

Although it is not part of the hierarchy for applet or frame windows, there is one other type of window that you will find valuable: Canvas. Canvas encapsulates a blank window upon which you can draw.

### **5.3. Event Handling**

There are several types of events. The most commonly handled events are those generated by the mouse, the keyboard, and various controls, such as a push button. Events are supported by the java.awt.event package.

#### **5.3.1. The Delegation Event Model**

The modern approach to handling events is based on the delegation event model, which defines standard and consistent mechanisms to generate and process events. Its concept is quite simple: a source generates an event and sends it to one or more listeners. In this scheme, the listener simply waits until it receives an event. Once received, the listener processes the event and then returns. The advantage of this design is that the application logic that processes events is cleanly separated from the user interface logic that generates those events. A user interface element is

able to "delegate" the processing of an event to a separate piece of code. In the delegation event model, listeners must register with a source in order to receive an event notification. This provides an important benefit: notifications are sent only to listeners that want to receive them. Previously, an event was propagated up the containment hierarchy until it was handled by a component. This required components to receive events that they did not process, and it wasted valuable time. The delegation event model eliminates this overhead.

Note Java also allows you to process events without using the delegation event model. This can be done by extending an AWT component. However, the delegation event model is the preferred design for the reasons just cited. The following sections define events and describe the roles of sources and listeners.

## **Events**

In the delegation model, an event is an object that describes a state change in a source. It can be generated as a consequence of a person interacting with the elements in a graphical user interface. Some of the activities that cause events to be generated are pressing a button, entering a character via the keyboard, selecting an item in a list, and clicking the mouse. Many other user operations could also be cited as examples. Events may also occur that are not directly caused by interactions with a user interface. For example, an event may be generated when a timer expires, a counter exceeds a value, a software or hardware failure occurs, or an operation is completed. You are free to define events that are appropriate for your application.

## **Event Sources**

A source is an object that generates an event. This occurs when the internal state of that object changes in some way. Sources may generate more than one type of event. A source must register listeners in order for the listeners to receive notifications about a specific type of event. Each type of event has its own registration method. Here is the general form:

*public void addTypeListener(TypeListener el)*

Here, Type is the name of the event and el is a reference to the event listener. For example, the method that registers a keyboard event listener is called `addKeyListener()`. The method that registers a mouse motion listener is called `addMouseMotionListener()`. When an event occurs, all registered listeners are notified and receive a copy of the event object. This is known as multicasting the event. In all cases, notifications are sent only to listeners that register to receive them. Some sources may allow only one listener to register. The general form of such a method



is this:

```
public void addTypeListener(TypeListener el) throws java.util.TooManyListenersException
```

Here, Type is the name of the event and el is a reference to the event listener. When such an event occurs, the registered listener is notified. This is known as unicasting the event. A source must also provide a method that allows a listener to unregister an interest in a specific type of event. The general form of such a method is this:

```
public void removeTypeListener(TypeListener el)
```

Here, Type is the name of the event and el is a reference to the event listener. For example, to remove a keyboard listener, you would call `removeKeyListener( )`. The methods that add or remove listeners are provided by the source that generates events. For example, the Component class provides methods to add and remove keyboard and mouse event listeners.

### **Event Listeners**

A listener is an object that is notified when an event occurs. It has two major requirements. First, it must have been registered with one or more sources to receive notifications about specific types of events. Second, it must implement methods to receive and process these notifications.

The methods that receive and process events are defined in a set of interfaces found in `java.awt.event`. For example, the `MouseMotionListener` interface defines two methods to receive notifications when the mouse is dragged or moved. Any object may receive and process one or both of these events if it provides an implementation of this interface. Many other listener interfaces are discussed later in this and other chapters.

### **Event Classes**

The classes that represent events are at the core of Java's event handling mechanism. Thus, we begin our study of event handling with a tour of the event classes. As you will see, they provide a consistent, easy-to-use means of encapsulating events.

At the root of the Java event class hierarchy is `EventObject`, which is in `java.util`. It is the super class for all events. Its one constructor is shown here:

```
EventObject(Object src)
```

Here, src is the object that generates this event. `EventObject` contains two methods: `getSource( )` and `toString( )`. The `getSource( )` method returns the source of the event. Its general form is shown here:

```
Object getSource( )
```

As expected, `toString()` returns the string equivalent of the event. The class `AWTEvent`, defined within the `java.awt` package, is a subclass of `EventObject`. It is the superclass (either directly or indirectly) of all AWT-based events used by the delegation event model. Its `getID()` method can be used to determine the type of the event. The signature of this method is shown here:

*int getID()*

`EventObject` is a superclass of all events. `AWTEvent` is a superclass of all AWT events that are handled by the delegation event model.

The package `java.awt.event` defines several types of events that are generated by various user interface elements. The most commonly used constructors and methods in each class are described in the following sections.

- `ActionEvent` Generated when a button is pressed, a list item is double-clicked, or a menu item is selected.
- `AdjustmentEvent` Generated when a scroll bar is manipulated.
- `ComponentEvent` Generated when a component is hidden, moved, resized, or becomes visible.
- `ContainerEvent` Generated when a component is added to or removed from a container.
- `FocusEvent` Generated when a component gains or loses keyboard focus.
- `InputEvent` Abstract super class for all component input event classes.
- `ItemEvent` Generated when a check box or list item is clicked; also occurs when a choice selection is made or a checkable menu item is selected or deselected.
- `KeyEvent` Generated when input is received from the keyboard.
- `MouseEvent` Generated when the mouse is dragged, moved, clicked, pressed, or released; also generated when the mouse enters or exits a component.
- `TextEvent` Generated when the value of a text area or text field is changed.
- `WindowEvent` Generated when a window is activated, closed, deactivated, deiconified, iconified, opened, or quit.

## **5.4. Working with Graphics**

The AWT supports a rich assortment of graphics methods. All graphics are drawn relative to a window. This can be the main window of an applet, a child window of an applet, or a stand-alone application window. The origin of each window is at the top-left corner and is 0,0.

Coordinates are specified in pixels. All output to a window takes place through a graphics context. A graphics context is encapsulated by the Graphics class and is obtained in two ways:

1. It is passed to an applet when one of its various methods, such as `paint()` or `update()`, is called.
2. It is returned by the `getGraphics()` method of Component.

For the remainder of the examples in this chapter, we will be demonstrating graphics in the main applet window. However, the same techniques will apply to any other window. The Graphics class defines a number of drawing functions. Each shape can be drawn edge-only or filled. Objects are drawn and filled in the currently selected graphics color, which is black by default. When a graphics object is drawn that exceeds the dimensions of the window, output is automatically clipped. Let's take a look at several of the drawing methods.

#### **5.4.1. Drawing Lines**

Lines are drawn by means of the `drawLine()` method, shown here:

```
void drawLine(int startX, int startY, int endX, int endY)
```

`drawLine()` displays a line in the current drawing color that begins at `startX, startY` and ends at `endX, endY`.

The following applet draws several lines:

```
// Draw lines  
import java.awt.*;  
import java.applet.*;  
/*  
<applet code="Lines" width=300 height=200>  
</applet>  
*/  
public class Lines extends Applet {  
public void paint(Graphics g) {  
g.drawLine(0, 0, 100, 100);  
g.drawLine(0, 100, 100, 0);  
g.drawLine(40, 25, 250, 180);  
g.drawLine(75, 90, 400, 400);  
g.drawLine(20, 150, 400, 40);
```

```
g.drawLine(5, 290, 80, 19);  
}  
}
```

## 5.5. Using AWT Controls

This chapter continues our exploration of the Abstract Window Toolkit (AWT). It examines the standard controls and layout managers defined by Java. Controls are components that allow a user to interact with your application in various ways—for example, a commonly used control is the push button.

The AWT supports the following types of controls: Labels, Push buttons, Check boxes, Choice lists, Lists, Scroll bars, Text editing. These controls are subclasses of `Component`.

### 5.5.1. Adding and Removing Controls

To include a control in a window, you must add it to the window. To do this, you must first create an instance of the desired control and then add it to a window by calling `add()`, which is defined by `Container`. The `add()` method has several forms. The following form is the one that is used for the first part of this chapter:

```
Component add(Component compObj)
```

Here, `compObj` is an instance of the control that you want to add. A reference to `compObj` is returned. Once a control has been added, it will automatically be visible whenever its parent window is displayed.

Sometimes you will want to remove a control from a window when the control is no longer needed. To do this, call `remove()`. This method is also defined by `Container`. It has this general form:

```
void remove(Component obj)
```

Here, `obj` is a reference to the control you want to remove. You can remove all controls by calling `removeAll()`.

### 5.5.2. Responding to Controls

Except for labels, which are passive controls, all controls generate events when they are accessed by the user. For example, when the user clicks on a push button, an event is sent that identifies the push button. In general, your program simply implements the appropriate interface and then registers an event listener for each control that you need to monitor. As explained in Chapter 20,

once a listener has been installed, events are automatically sent to it. In the sections that follow, the appropriate interface for each control is specified.

### 5.5.3. Labels

The easiest control to use is a label. A label is an object of type `Label`, and it contains a string, which it displays. Labels are passive controls that do not support any interaction with the user. `Label` defines the following constructors:

*`Label( )`*

*`Label(String str)`*

*`Label(String str, int how)`*

The first version creates a blank label. The second version creates a label that contains the string specified by `str`. This string is left-justified. The third version creates a label that contains the string specified by `str` using the alignment specified by `how`. The value of `how` must be one of these three constants: `Label.LEFT`, `Label.RIGHT`, or `Label.CENTER`.

The following example creates three labels and adds them to an applet:

```
// Demonstrate Labels
import java.awt.*;
import java.applet.*;
/*
<applet code="LabelDemo" width=300 height=200>
</applet>
*/
public class LabelDemo extends Applet {
    public void init() {
        Label one = new Label("One");
        Label two = new Label("Two");
        Label three = new Label("Three");
        add(one);
        add(two);
        add(three);
    }
}
```

#### 5.5.4. Buttons

The most widely used control is the push button. A push button is a component that contains a label and that generates an event when it is pressed. Push buttons are objects of type `Button`. `Button` defines these two constructors:

*`Button( )`*

*`Button(String str)`*

The first version creates an empty button. The second creates a button that contains `str` as a label.

#### Handling Buttons

Each time a button is pressed, an action event is generated. This is sent to any listeners that previously registered an interest in receiving action event notifications from that component. Each listener implements the `ActionListener` interface. That interface defines the `actionPerformed( )` method, which is called when an event occurs. An `ActionEvent` object is supplied as the argument to this method. It contains both a reference to the button that generated the event and a reference to the string that is the label of the button. Usually, either value may be used to identify the button, as you will see.

Here is an example that creates three buttons labeled "Yes," "No," and "Undecided." Each time one is pressed, a message is displayed that reports which button has been pressed. In this version, the label of the button is used to determine which button has been pressed. The label is obtained by calling the `getActionCommand( )` method on the `ActionEvent` object passed to `actionPerformed( )`.

```
// Demonstrate Buttons
```

```
import java.awt.*;
```

```
import java.awt.event.*;
```

```
import java.applet.*;
```

```
/*
```

```
<applet code="ButtonDemo" width=250 height=150>
```

```
</applet>
```

```
*/
```

```
public class ButtonDemo extends Applet implements ActionListener
```

```
{
```

```
String msg = "";
```

```

Button yes, no, maybe;
public void init() {
    yes = new Button("Yes");
    no = new Button("No");
    maybe = new Button("Undecided");
    add(yes);
    add(no);
    add(maybe);
}
yes.addActionListener(this);
no.addActionListener(this);
maybe.addActionListener(this);
public void actionPerformed(ActionEvent ae) {
    String str = ae.getActionCommand();
    if(str.equals("Yes")) {
        msg = "You pressed Yes.";
    }
    else if(str.equals("No")) {
        msg = "You pressed No.";
    }
    else {
        msg = "You pressed Undecided.";
    }
    repaint();
}
public void paint(Graphics g) {
    g.drawString(msg, 6, 100);
}
}

```

#### **5.5.5. Check Boxes**

A check box is a control that is used to turn an option on or off. It consists of a small box that can either contain a check mark or not. There is a label associated with each check box that describes what option the box represents. You change the state of a check box by clicking on it. Check boxes can be used individually or as part of a group. Check boxes are objects of the `Checkbox` class.

`Checkbox` supports these constructors:

*`Checkbox( )`*

*`Checkbox(String str)`*

*`Checkbox(String str, boolean on)`*

*`Checkbox(String str, boolean on, CheckboxGroup cbGroup)`*

*`Checkbox(String str, CheckboxGroup cbGroup, boolean on)`*

The first form creates a check box whose label is initially blank. The state of the check box is unchecked. The second form creates a check box whose label is specified by `str`. The state of the check box is unchecked. The third form allows you to set the initial state of the check box. If `on` is true, the check box is initially checked; otherwise, it is cleared. The fourth and fifth forms create a check box whose label is specified by `str` and whose group is specified by `cbGroup`. If this check box is not part of a group, then `cbGroup` must be null. (Check box groups are described in the next section.) The value of `on` determines the initial state of the check box.

### **Handling Check Boxes**

Each time a check box is selected or deselected, an item event is generated. This is sent to any listeners that previously registered an interest in receiving item event notifications from that component. Each listener implements the `ItemListener` interface. That interface defines the `itemStateChanged( )` method. An `ItemEvent` object is supplied as the argument to this method. It contains information about the event (for example, whether it was a selection or deselection).

The following program creates four check boxes. The initial state of the first box is checked. The status of each check box is displayed. Each time you change the state of a check box, the status display is updated.

```
// Demonstrate check boxes.
```

```
import java.awt.*;
```

```
import java.awt.event.*;
```

```
import java.applet.*;
```



```

/*
<applet code="CheckboxDemo" width=250 height=200>
</applet>
*/

public class CheckboxDemo extends Applet implements ItemListener
{
    String msg = "";
    Checkbox Win98, winNT, solaris, mac;

    public void init() {
        Win98 = new Checkbox("Windows 98", null, true);
        winNT = new Checkbox("Windows NT");
        solaris = new Checkbox("Solaris");
        mac = new Checkbox("MacOS");
        add(Win98);
        add(winNT);
        add(solaris);
        add(mac);
        Win98.addItemListener(this);
        winNT.addItemListener(this);
        solaris.addItemListener(this);
    }
    mac.addItemListener(this);

    public void itemStateChanged(ItemEvent ie) {
        repaint();
    }

    // Display current state of the check boxes.
    public void paint(Graphics g) {
        msg = "Current state: ";
        g.drawString(msg, 6, 80);
        msg = " Windows 98: " + Win98.getState();
        g.drawString(msg, 6, 100);
    }
}

```

```

msg = " Windows NT: " + winNT.getState();
g.drawString(msg, 6, 120);
msg = " Solaris: " + solaris.getState();
g.drawString(msg, 6, 140);
msg = " MacOS: " + mac.getState();
g.drawString(msg, 6, 160);
}
}

```

### 5.5.6. CheckboxGroup

It is possible to create a set of mutually exclusive check boxes in which one and only one check box in the group can be checked at any one time. These check boxes are often called radio buttons, because they act like the station selector on a car radio—only one station can be selected at any one time. To create a set of mutually exclusive check boxes, you must first define the group to which they will belong and then specify that group when you construct the check boxes. Check box groups are objects of type `CheckboxGroup`. Only the default constructor is defined, which creates an empty group.

You can determine which check box in a group is currently selected by calling `getSelectedCheckbox()`. You can set a check box by calling `setSelectedCheckbox()`.

These methods are as follows:

*Checkbox* `getSelectedCheckbox()`

*void* `setSelectedCheckbox(Checkbox which)`

Here, `which` is the check box that you want to be selected. The previously selected check box will be turned off. Here is a program that uses check boxes that are part of a group:

```

// Demonstrate check box group.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="CBGroup" width=250 height=200>
</applet>
*/

```

```

public class CBGroup extends Applet implements ItemListener {
    String msg = "";
    Checkbox Win98, winNT, solaris, mac;
    CheckboxGroup cbg;
    public void init() {
        cbg = new CheckboxGroup();
        Win98 = new Checkbox("Windows 98", cbg, true);
        winNT = new Checkbox("Windows NT", cbg, false);
        solaris = new Checkbox("Solaris", cbg, false);
        mac = new Checkbox("MacOS", cbg, false);
        add(Win98);
        add(winNT);
        add(solaris);
        add(mac);
        Win98.addItemListener(this);
    }
    winNT.addItemListener(this);
    solaris.addItemListener(this);
    mac.addItemListener(this);
    public void itemStateChanged(ItemEvent ie) {
        repaint();
    }
    // Display current state of the check boxes.
    public void paint(Graphics g) {
        msg = "Current selection: ";
        msg += cbg.getSelectedCheckbox().getLabel();
        g.drawString(msg, 6, 100);
    }
}

```

### 5.5.7. Choice Controls

The Choice class is used to create a pop-up list of items from which the user may choose. Thus, a Choice control is a form of menu. When inactive, a Choice component takes up only enough space to show the currently selected item. When the user clicks on it, the whole list of choices pops up, and a new selection can be made. Each item in the list is a string that appears as a left-justified label in the order it is added to the Choice object. Choice only defines the default constructor, which creates an empty list. To add a selection to the list, call addItem( ) or add( ). They have these general forms:

```
void addItem(String name)
```

```
void add(String name)
```

Here, name is the name of the item being added. Items are added to the list in the order in which calls to add( ) or addItem( ) occur.

To determine which item is currently selected, you may call either getSelectedItem( ) or getSelectedIndex( ). These methods are shown here:

```
String getSelectedItem( )
```

```
int getSelectedIndex( )
```

The getSelectedItem( ) method returns a string containing the name of the item. getSelectedIndex( ) returns the index of the item. The first item is at index 0. By default, the first item added to the list is selected. To obtain the number of items in the list, call getItemCount( ).

You can set the currently

selected item using the select( ) method with either a zero-based integer index or a string that will match a name in the list. These methods are shown here:

```
int getItemCount( )
```

```
void select(int index)
```

```
void select(String name)
```

Given an index, you can obtain the name associated with the item at that index by calling getItem( ), which has this general form:

```
String getItem(int index)
```

Here, index specifies the index of the desired item.

### **Handling Choice Lists**

Each time a choice is selected, an item event is generated. This is sent to any listeners that previously registered an interest in receiving item event notifications from that component. Each

listener implements the ItemListener interface. That interface defines the itemStateChanged( ) method. An ItemEvent object is supplied as the argument to this method. Here is an example that creates two Choice menus. One selects the operating system.

The other selects the browser.

```
// Demonstrate Choice lists.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="ChoiceDemo" width=300 height=180>
</applet>
*/
public class ChoiceDemo extends Applet implements ItemListener {
    Choice os, browser;
    String msg = "";
    public void init() {
        os = new Choice();
        browser = new Choice();
        // add items to os list
        os.add("Windows 98");
        os.add("Windows NT");
        os.add("Solaris");
        os.add("MacOS");
        // add items to browser list
        browser.add("Netscape 1.1");
        browser.add("Netscape 2.x");
        browser.add("Netscape 3.x");
        browser.add("Netscape 4.x");
        browser.add("Internet Explorer 2.0");
        browser.add("Internet Explorer 3.0");
        browser.add("Internet Explorer 4.0");
```

```

browser.add("Lynx 2.4");
browser.select("Netscape 4.x");
// add choice lists to window
add(os);
add(browser);
}
// register to receive item events
os.addItemListener(this);
browser.addItemListener(this);
public void itemStateChanged(ItemEvent ie) {
repaint();
}
// Display current selections.
public void paint(Graphics g) {
msg = "Current OS: ";
msg += os.getSelectedItem();
g.drawString(msg, 6, 120);
msg = "Current Browser: ";
msg += browser.getSelectedItem();
g.drawString(msg, 6, 140);
}
}

```

### 5.5.8. Lists

The List class provides a compact, multiple-choice, scrolling selection list. Unlike the Choice object, which shows only the single selected item in the menu, a List object can be constructed to show any number of choices in the visible window. It can also be created to allow multiple selections. List provides these constructors:

*List( )*

*List(int numRows)*

*List(int numRows, boolean multipleSelect)*

The first version creates a List control that allows only one item to be selected at any one time. In the second form, the value of numRows specifies the number of entries in the list that will always be visible (others can be scrolled into view as needed). In the third form, if multipleSelect is true, then the user may select two or more items at a time. If it is false, then only one item may be selected.

To add a selection to the list, call add( ). It has the following two forms:

```
void add(String name)
```

```
void add(String name, int index)
```

Here, name is the name of the item added to the list. The first form adds items to the end of the list. The second form adds the item at the index specified by index. Indexing begins at zero. You can specify -1 to add the item to the end of the list. For lists that allow only single selection, you can determine which item is currently selected by calling either getSelectedItem( ) or getSelectedIndex(). These methods are shown here:

```
String getSelectedItem( )
```

```
int getSelectedIndex( )
```

The getSelectedItem( ) method returns a string containing the name of the item. If more than one item is selected or if no selection has yet been made, null is returned. getSelectedIndex( ) returns the index of the item. The first item is at index 0. If more than one item is selected, or if no selection has yet been made, -1 is returned. For lists that allow multiple selection, you must use either getSelectedItems( ) or getSelectedIndexes( ), shown here, to determine the current selections:

```
String[ ] getSelectedItems( )
```

```
int[ ] getSelectedIndexes( )
```

getSelectedItems( ) returns an array containing the names of the currently selected items.

getSelectedIndexes( ) returns an array containing the indexes of the currently selected items.

To obtain the number of items in the list, call getItemCount( ). You can set the currently selected item by using the select( ) method with a zero-based integer index. These methods are shown here:

```
int getItemCount( )
```

```
void select(int index)
```

Given an index, you can obtain the name associated with the item at that index by calling `getItem( )`, which has this general form:

```
String getItem(int index)
```

Here, index specifies the index of the desired item.

### **Handling Lists**

To process list events, you will need to implement the `ActionListener` interface. Each time a List item is double-clicked, an `ActionEvent` object is generated. Its `getActionCommand( )` method can be used to retrieve the name of the newly selected item. Also, each time an item is selected or deselected with a single click, an `ItemEvent` object is generated. Its `getStateChange( )` method can be used to determine whether a selection or deselection triggered this event. `getItemSelectable( )` returns a reference to the object that triggered this event.

Here is an example that converts the Choice controls in the preceding section into List components, one multiple choice and the other single choice:

```
// Demonstrate Lists.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="ListDemo" width=300 height=180>
</applet>
*/
public class ListDemo extends Applet implements ActionListener {
    List os, browser;
    String msg = "";
    public void init() {
        os = new List(4, true);
        browser = new List(4, false);
        // add items to os list
        os.add("Windows 98");
        os.add("Windows NT");
        os.add("Solaris");
```



```

os.add("MacOS");
// add items to browser list
browser.add("Netscape 1.1");
browser.add("Netscape 2.x");
browser.add("Netscape 3.x");
browser.add("Netscape 4.x");
browser.add("Internet Explorer 2.0");
browser.add("Internet Explorer 3.0");
browser.add("Internet Explorer 4.0");
browser.add("Lynx 2.4");
browser.select(1);
// add lists to window
add(os);
add(browser);
}
// register to receive action events
os.addActionListener(this);
browser.addActionListener(this);
public void actionPerformed(ActionEvent ae) {
repaint();
}
// Display current selections.
public void paint(Graphics g) {
int idx[];
msg = "Current OS: ";
idx = os.getSelectedIndexes();
for(int i=0; i<idx.length; i++)
msg += os.getItem(idx[i]) + " ";
g.drawString(msg, 6, 120);
msg = "Current Browser: ";
msg += browser.getSelectedItem();

```

```
g.drawString(msg, 6, 140);  
}  
}
```

### 5.5.9. Scroll Bars

Scroll bars are used to select continuous values between a specified minimum and maximum. Scroll bars may be oriented horizontally or vertically. A scroll bar is actually a composite of several individual parts. Each end has an arrow that you can click to move the current value of the scroll bar one unit in the direction of the arrow. The current value of the scroll bar relative to its minimum and maximum values is indicated by the slider box (or thumb) for the scroll bar. The slider box can be dragged by the user to a new position. The scroll bar will then reflect this value. In the background space on either side of the thumb, the user can click to cause the thumb to jump in that direction by some increment larger than 1. Typically, this action translates into some form of page up and page down. Scroll bars are encapsulated by the Scrollbar class.

Scrollbar defines the following constructors:

*Scrollbar( )*

*Scrollbar(int style)*

*Scrollbar(int style, int initialValue, int thumbSize, int min, int max)*

The first form creates a vertical scroll bar. The second and third forms allow you to specify the orientation of the scroll bar. If style is Scrollbar.VERTICAL, a vertical scroll bar is created. If style is Scrollbar.HORIZONTAL, the scroll bar is horizontal. In the third form of the constructor, the initial value of the scroll bar is passed in initialValue. The number of units represented by the height of the thumb is passed in thumbSize. The minimum and maximum values for the scroll bar are specified by min and max. If you construct a scroll bar by using one of the first two constructors, then you need to set its parameters by using setValues( ), shown here, before it can be used:

*void setValues(int initialValue, int thumbSize, int min, int max)*

The parameters have the same meaning as they have in the third constructor just described. To obtain the current value of the scroll bar, call getValue( ). It returns the current setting. To set the current value, call setValue( ). These methods are as follows:

*int getValue( )*

*void setValue(int newValue)*

Here, `newValue` specifies the new value for the scroll bar. When you set a value, the slider box inside the scroll bar will be positioned to reflect the new value. You can also retrieve the minimum and maximum values via `getMinimum()` and `getMaximum()`, shown here:

```
int getMinimum()
```

```
int getMaximum()
```

They return the requested quantity. By default, 1 is the increment added to or subtracted from the scroll bar each time it is scrolled up or down one line. You can change this increment by calling `setUnitIncrement()`. By default, page-up and page-down increments are 10. You can change this value by calling `setBlockIncrement()`. These methods are shown here:

```
void setUnitIncrement(int newIncr)
```

```
void setBlockIncrement(int newIncr)
```

### **Handling Scroll Bars**

To process scroll bar events, you need to implement the `AdjustmentListener` interface. Each time a user interacts with a scroll bar, an `AdjustmentEvent` object is generated. Its `getAdjustmentType()` method can be used to determine the type of the adjustment. The types of adjustment events are as follows:

`BLOCK_DECREMENT` A page-down event has been generated.

`BLOCK_INCREMENT` A page-up event has been generated.

`TRACK` An absolute tracking event has been generated.

`UNIT_DECREMENT` The line-down button in a scroll bar has been pressed.

`UNIT_INCREMENT` The line-up button in a scroll bar has been pressed.

The following example creates both a vertical and a horizontal scroll bar. The current settings of the scroll bars are displayed. If you drag the mouse while inside the window, the coordinates of each drag event are used to update the scroll bars. An asterisk is displayed at the current drag position.

#### **5.5.10. TextField**

The `TextField` class implements a single-line text-entry area, usually called an edit control. Text fields allow the user to enter strings and to edit the text using the arrow keys, cut and paste keys, and mouse selections. `TextField` is a subclass of `TextComponent`. `TextField` defines the following constructors:

*TextField( )*

*TextField(int numChars)*

*TextField(String str)*

*TextField(String str, int numChars)*

The first version creates a default text field. The second form creates a text field that is numChars characters wide. The third form initializes the text field with the string contained in str. The fourth form initializes a text field and sets its width.

TextField (and its superclass TextComponent) provides several methods that allow you to utilize a text field. To obtain the string currently contained in the text field, call getText(). To set the text, call setText( ). These methods are as follows:

*String getText( )*

*void setText(String str)*

*Here, str is the new string.*

The user can select a portion of the text in a text field. Also, you can select a portion of text under program control by using select( ). Your program can obtain the currently selected text by calling getSelectedText( ). These methods are shown here:

*String getSelectedText( )*

*void select(int startIndex, int endIndex)*

getSelectedText( ) returns the selected text. The select( ) method selects the characters beginning at startIndex and ending at endIndex-1. You can control whether the contents of a text field may be modified by the user by calling setEditable( ). You can determine editability by calling isEditable( ). These methods are shown here:

*boolean isEditable( )*

*void setEditable(boolean canEdit)*

isEditable( ) returns true if the text may be changed and false if not. In setEditable( ), if canEdit is true, the text may be changed. If it is false, the text cannot be altered. There may be times when you will want the user to enter text that is not displayed, such as a password. You can disable the echoing of the characters as they are typed by calling setEchoChar( ). This method specifies a single character that the TextField will display when characters are entered (thus, the actual characters typed will not be shown). You can check a text field to see if it is in this mode with the echoCharIsSet( )

method. You can retrieve the echo character by calling the `getEchoChar( )` method. These methods are as follows:

```
void setEchoChar(char ch)
```

```
boolean echoCharIsSet( )
```

```
char getEchoChar( )
```

Here, `ch` specifies the character to be echoed.

### **Handling a TextField**

Since text fields perform their own editing functions, your program generally will not respond to individual key events that occur within a text field. However, you may want to respond when the user presses ENTER. When this occurs, an action event is generated. Here is an example that creates the classic user name and password screen:

```
// Demonstrate text field.  
import java.awt.*;  
import java.awt.event.*;  
import java.applet.*;  
/*  
<applet code="TextFieldDemo" width=380 height=150>  
</applet>  
*/  
  
public class TextFieldDemo extends Applet  
implements ActionListener {  
    TextField name, pass;  
  
    public void init() {  
        Label namep = new Label("Name: ", Label.RIGHT);  
        Label passp = new Label("Password: ", Label.RIGHT);  
        name = new TextField(12);  
        pass = new TextField(8);  
        pass.setEchoChar('?');  
        add(namep);  
        add(name);  
        add(passp);
```

```

    add(pass);
- 516 -
}
// register to receive action events
name.addActionListener(this);
pass.addActionListener(this);
// User pressed Enter.
public void actionPerformed(ActionEvent ae) {
    repaint();
}
public void paint(Graphics g) {
    g.drawString("Name: " + name.getText(), 6, 60);
    g.drawString("Selected text in name: "
+ name.getSelectedText(), 6, 80);
    g.drawString("Password: " + pass.getText(), 6, 100);
}
}

```

### 5.5.11. Using a TextArea

Sometimes a single line of text input is not enough for a given task. To handle these situations, the AWT includes a simple multiline editor called `TextArea`. Following are the constructors for `TextArea`:

```
TextArea( )
```

```
TextArea(int numLines, int numChars)
```

```
TextArea(String str)
```

```
TextArea(String str, int numLines, int numChars)
```

```
TextArea(String str, int numLines, int numChars, int sBars)
```

Here, `numLines` specifies the height, in lines, of the text area, and `numChars` specifies its width, in characters. Initial text can be specified by `str`. In the fifth form you can specify the scroll bars that you want the control to have. `sBars` must be one of these values:

`SCROLLBARS_BOTH`

`SCROLLBARS_NONE`

SCROLLBARS\_HORIZONTAL\_ONLY SCROLLBARS\_VERTICAL\_ONLY

TextArea is a subclass of TextComponent. Therefore, it supports the `getText( )`, `setText( )`, `getSelectedText( )`, `select( )`, `isEditable( )`, and `setEditable( )` methods described in the preceding section.

TextArea adds the following methods:

*void append(String str)*

*void insert(String str, int index)*

The `append( )` method appends the string specified by `str` to the end of the current text. `insert( )` inserts the string passed in `str` at the specified index. To replace text, call `replaceRange( )`. It replaces the characters from `startIndex` to `endIndex-1`, with the replacement text passed in `str`.

## **5.6. Layout Managers**

All of the components that we have shown so far have been positioned by the default layout manager. Layout manager automatically arranges your controls within a window by using some type of algorithm. If you have programmed for other GUI environments, such as Windows, then you are controls by hand, too, you generally won't want to, for two main reasons. First, it is very tedious to manually lay out a large number of components. Second, sometimes the width and height information is not yet available when you need to arrange some control, because the native toolkit components haven't been realized. This is a chicken-and-egg situation; it is pretty confusing to figure out when it is okay to use the size of a given component to position it relative to another.

Each Container object has a layout manager associated with it. A layout manager is an instance of any class that implements the `LayoutManager` interface. The layout manager is set by the `setLayout( )` method. If no call to `setLayout( )` is made, then the default layout manager is used. Whenever a container is resized (or sized for the first time), the layout manager is used to position each of the components within it.

The `setLayout( )` method has the following general form:

*void setLayout(LayoutManager layoutObj)*

Here, `layoutObj` is a reference to the desired layout manager. If you wish to disable the layout manager and position components manually, pass null for `layoutObj`. If you do this, you will need to determine the shape and position of each component manually, using the `setBounds( )` method defined by `Component`. Normally, you will want to use a layout manager.

Each layout manager keeps track of a list of components that are stored by their names. The layout manager is notified each time you add a component to a container. Whenever the container needs to be resized, the layout manager is consulted via its `minimumLayoutSize()` and `preferredLayoutSize()` methods. Each component that is being managed by a layout manager contains the `getPreferredSize()` and `getMinimumSize()` methods. These return the preferred and minimum size required to display each component. The layout manager will honor these requests if at all possible, while maintaining the integrity of the layout policy. You may override these methods for controls that you subclass. Default values are provided otherwise. Java has several predefined `LayoutManager` classes, which are described next. You can use the layout manager that best fits your application.

### **5.6.1. FlowLayout**

`FlowLayout` is the default layout manager. This is the layout manager that the preceding examples have used. `FlowLayout` implements a simple layout style, which is similar to how words flow in a text editor. Components are laid out from the upper-left corner, left to right and top to bottom. When no more components fit on a line, the next one appears on the next line. A small space is left between each component, above and below, as well as left and right. Here are the constructors for `FlowLayout`:

*`FlowLayout()`*

*`FlowLayout(int how)`*

*`FlowLayout(int how, int horz, int vert)`*

The first form creates the default layout, which centers components and leaves five pixels of space between each component. The second form lets you specify how each line is aligned.

Valid values for `how` are as follows:

`FlowLayout.LEFT`

`FlowLayout.CENTER`

`FlowLayout.RIGHT`

These values specify left, center, and right alignment, respectively. The third form allows you to specify the horizontal and vertical space left between components in `horz` and `vert`, respectively.

Here is a version of the `CheckboxDemo` applet shown earlier in this chapter, modified so that it uses left-aligned flow layout.

```
// Use left-aligned flow layout.
```



```

import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="FlowLayoutDemo" width=250 height=200>
</applet>
*/

public class FlowLayoutDemo extends Applet
implements ItemListener {
String msg = "";
Checkbox Win98, winNT, solaris, mac;

public void init() {
// set left-aligned flow layout
setLayout(new FlowLayout(FlowLayout.LEFT));
Win98 = new Checkbox("Windows 98", null, true);
winNT = new Checkbox("Windows NT");
solaris = new Checkbox("Solaris");
mac = new Checkbox("MacOS");
add(Win98);
add(winNT);
add(solaris);
add(mac);
}

// register to receive item events
Win98.addItemListener(this);
winNT.addItemListener(this);
solaris.addItemListener(this);
mac.addItemListener(this);

// Repaint when status of a check box changes.
public void itemStateChanged(ItemEvent ie) {
repaint();
}

```

```

}
// Display current state of the check boxes.
public void paint(Graphics g) {
    msg = "Current state: ";
    g.drawString(msg, 6, 80);
    msg = " Windows 98: " + Win98.getState();
    g.drawString(msg, 6, 100);
    msg = " Windows NT: " + winNT.getState();
    g.drawString(msg, 6, 120);
    msg = " Solaris: " + solaris.getState();
    g.drawString(msg, 6, 140);
    msg = " Mac: " + mac.getState();
    g.drawString(msg, 6, 160);
}
}

```

### 5.6.2. BorderLayout

The BorderLayout class implements a common layout style for top-level windows. It has four narrow, fixed-width components at the edges and one large area in the center. The four sides are referred to as north, south, east, and west. The middle area is called the center. Here are the constructors defined by BorderLayout:

*BorderLayout( )*

*BorderLayout(int horz, int vert)*

The first form creates a default border layout. The second allows you to specify the horizontal and vertical space left between components in horz and vert, respectively. BorderLayout defines the following constants that specify the regions:

BorderLayout.CENTER BorderLayout.SOUTH

BorderLayout.EAST BorderLayout.WEST

BorderLayout.NORTH

When adding components, you will use these constants with the following form of add( ), which is defined by Container:

*void add(Component compObj, Object region);*

Here, compObj is the component to be added, and region specifies where the component will be added.

Here is an example of a BorderLayout with a component in each layout area:

```
// Demonstrate BorderLayout.

import java.awt.*;
import java.applet.*;
import java.util.*;

/*
<applet code="BorderLayoutDemo" width=400 height=200>
</applet>
*/

public class BorderLayoutDemo extends Applet {
    public void init() {
        setLayout(new BorderLayout());
        add(new Button("This is across the top."),
            BorderLayout.NORTH);
        add(new Label("The footer message might go here."),
            BorderLayout.SOUTH);
        add(new Button("Right"), BorderLayout.EAST);
        add(new Button("Left"), BorderLayout.WEST);
        String msg = "The reasonable man adapts " +
            "himself to the world;\n" +
            "the unreasonable one persists in " +
            "trying to adapt the world to himself.\n" +
            "Therefore all progress depends " +
            "on the unreasonable man.\n\n" +
            "George Bernard Shaw\n\n";
        add(new TextArea(msg), BorderLayout.CENTER);
    }
}
```

## 5.7. Java Database Connectivity

**JDBC** is a Java-based data access technology (Java Standard Edition platform) from Sun Microsystems. It is an acronym as it is unofficially referred to as **Java Database Connectivity**, with DB being universally recognized as the abbreviation for **database**. JDBC is Java application programming interface that allows the Java programmers to access database management system from Java code. It was developed by JavaSoft, a subsidiary of Sun Microsystems.

**Java Database Connectivity** in short called as JDBC. It is a java API which enables the java programs to execute SQL statements. It is an application programming interface that defines how a java programmer can access the database in tabular format from Java code using a set of standard interfaces and classes written in the Java programming language.

JDBC is consists of four Components: The JDBC API, JDBC Driver Manager, The JDBC Test Suite and JDBC-ODBC Bridge.

Generally all Relational Database Management System supports SQL and we all know that Java is platform independent, so JDBC makes it possible to write a single database application that can run on different platforms and interact with different Database Management Systems.

Java Database Connectivity is similar to Open Database Connectivity (ODBC) which is used for accessing and managing database, but the difference is that JDBC is designed specifically for Java programs, whereas ODBC is not depended upon any language.

In short JDBC helps the programmers to write java applications that manage these three programming activities:

1. It helps us to connect to a data source, like a database.
2. It helps us in sending queries and updating statements to the database and
3. Retrieving and processing the results received from the database in terms of answering to your query.

#### **5.7.1. Understanding JDBC architecture**

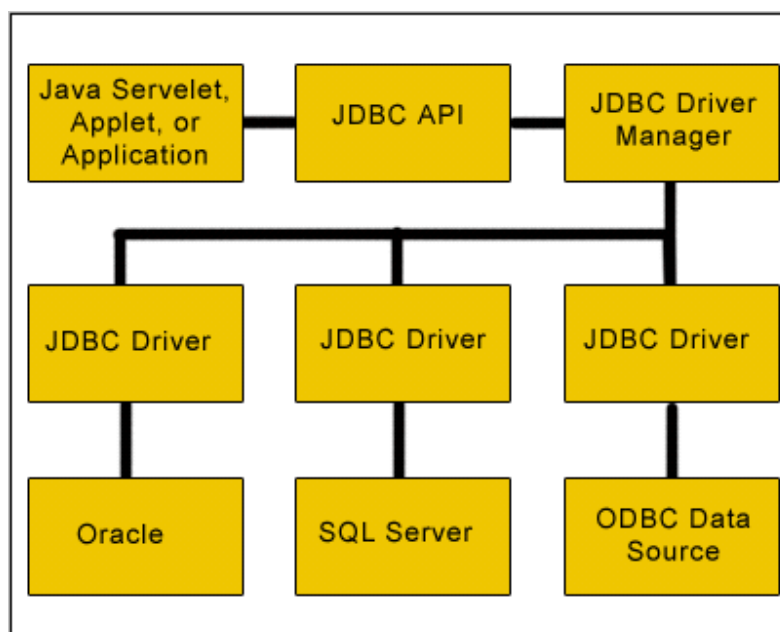
The primary function of the JDBC API is to provide a means for the developer to issue SQL statements and process the results in a consistent, database-independent manner. JDBC provides rich, object-oriented access to databases by defining classes and interfaces that represent objects such as:

- Database connections
- SQL statements

- Result Set
- Database metadata
- Prepared statements
- Binary Large Objects (BLOBs)
- Character Large Objects (CLOBs)
- Callable statements
- Database drivers
- Driver manager

The JDBC API uses a Driver Manager and database-specific drivers to provide transparent connectivity to heterogeneous databases. The JDBC driver manager ensures that the correct driver is used to access each data source. The Driver Manager is capable of supporting multiple concurrent drivers connected to multiple heterogeneous databases. The location of the driver manager with respect to the JDBC drivers and the servlet is shown in Figure.

### Layers of the JDBC Architecture

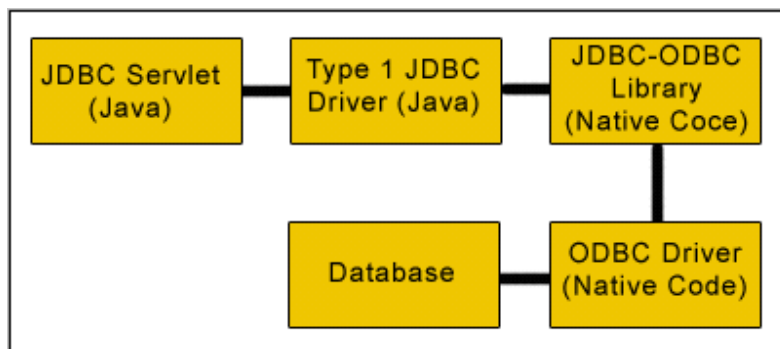


A **JDBC driver** translates standard *JDBC* calls into a network or database protocol or into a database library API call that facilitates communication with the database. This translation layer provides JDBC applications with database independence. If the back-end database changes, only

the JDBC driver need be replaced with few code modifications required. There are four distinct types of JDBC drivers.

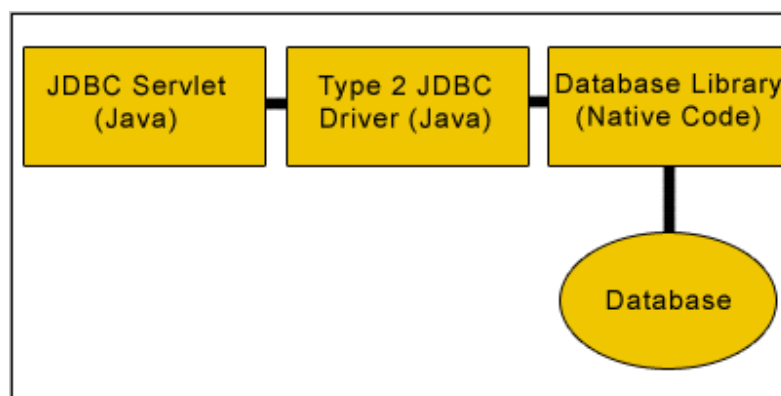
**Type 1 JDBC-ODBC Bridge.** Type 1 drivers act as a *"bridge"* between **JDBC** and another database connectivity mechanism such as **ODBC**. The **JDBC- ODBC** bridge provides JDBC access using most standard ODBC drivers. This driver is included in the Java 2 SDK within the **sun.jdbc.odbc** package. In this driver the java statements are converted to a jdbc statements. JDBC statements calls the ODBC by using the **JDBC-ODBC Bridge**. And finally the query is executed by the database. This driver has serious limitation for many applications.

### Type 1 JDBC Architecture



**Type 2 Java to Native API.** Type 2 drivers use the **Java Native Interface (JNI)** to make calls to a local database library API. This driver converts the JDBC calls into a database specific call for databases such as SQL, ORACLE etc. This driver communicates directly with the database server. It requires some native code to connect to the database. Type 2 drivers are usually faster than Type 1 drivers. Like Type 1 drivers, Type 2 drivers require native database client libraries to be installed and configured on the client machine.

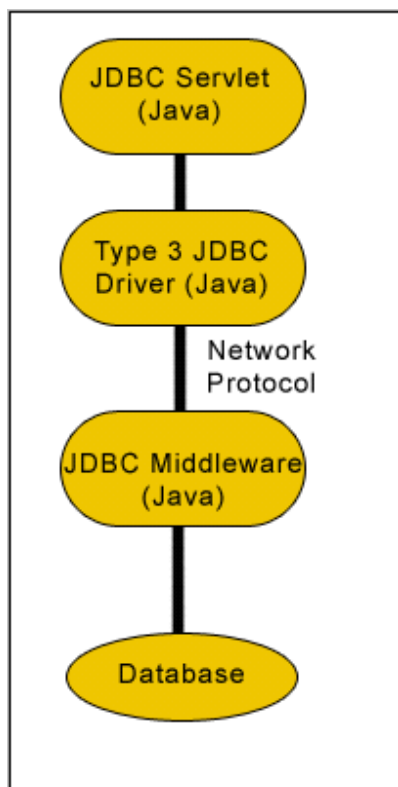
### Type 2 JDBC Architecture



**Type 3 Java to Network Protocol Or All- Java Driver.** Type 3 drivers are pure Java drivers that use a proprietary network protocol to communicate with JDBC middleware on the server. The middleware then translates the network protocol to database-specific function calls. Type 3 drivers are the most flexible JDBC solution because they do not require native database libraries on the client and can connect to many different databases on the back end. Type 3 drivers can be deployed over the Internet without client installation.

Java-----> JDBC statements-----> SQL statements -----> databases.

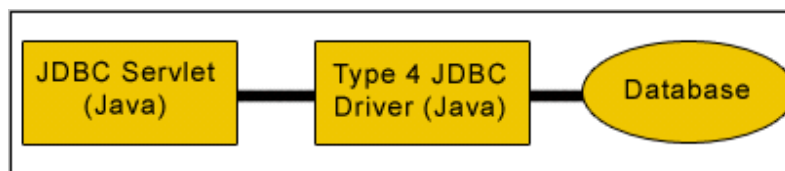
### **Type 3 JDBC Architecture**



**Type 4 Java to Database Protocol.** Type 4 drivers are pure Java drivers that implement a proprietary database protocol (like Oracle's SQL\*Net) to communicate directly with the

database. Like Type 3 drivers, they do not require native database libraries and can be deployed over the Internet without client installation. One drawback to Type 4 drivers is that they are database specific. Unlike Type 3 drivers, if your back-end database changes, you may have to purchase and deploy a new Type 4 driver (some Type 4 drivers are available free of charge from the database manufacturer). However, because Type 4 drivers communicate directly with the database engine rather than through middleware or a native library, they are usually the fastest JDBC drivers available. This driver directly converts the Java statements to SQL statements.

### **Type 4 JDBC Architecture**



So, you may be asking yourself, "Which is the right type of driver for your application?" Well, that depends on the requirements of your particular project. If you do not have the opportunity or inclination to install and configure software on each client, you can rule out Type 1 and Type 2 drivers.

However, if the cost of Type 3 or Type 4 drivers is prohibitive, Type 1 and type 2 drivers may become more attractive because they are usually available free of charge. Price aside, the debate will often boil down to whether to use Type 3 or Type 4 driver for a particular application. In this case, you may need to weigh the benefits of flexibility and interoperability against performance. Type 3 drivers offer your application the ability to transparently access different types of databases, while Type 4 drivers usually exhibit better performance and, like Type 1 and Type 2 drivers, may be available free of charge from the database manufacturer.

### **5.7.2. Working with JDBC**

Every JDBC program is made up of the following 4 steps:

- Open a connection to the DB
- Execute a SQL statement



- Process the result
- Close the connection to the DB

### 5.7.2.1. Opening a connection to the DB

There are two parts to this:

- loading a driver – we need a driver to allow our Java program to talk to the DB
- opening the connection itself

#### Loading the driver

The first step in using JDBC is to load a driver.

The method `Class.forName(String)` is used to load the JDBC driver class.

General syntax is

```
class.forName("Driver name");
```

Here are some examples:

The IBM DB2 driver:

```
Class.forName( "COM.ibm.db2.jdbc.app.DB2Driver" );
```

The SUN JDBC/ODBC Bridge driver:

```
Class.forName( "sun.jdbc.JdbcOdbcDriver" );
```

#### There are 4 categories of driver

##### Type 1 JDBC-ODBC Bridge (Native Code)

The JDBC type 1 driver, also known as the JDBC-ODBC bridge is a database driver implementation that employs the ODBC driver to connect to the database. The driver converts JDBC method calls into ODBC function calls. The bridge is usually used when there is no pure-Java driver available for a particular database.

##### Type 2 Native-API (Partly Java)

The JDBC type 2 driver, also known as the Native-API driver is a database driver implementation that uses the client-side libraries of the database. The driver converts JDBC method calls into native calls of the database API.

##### Type 3 Net-protocol (All Java)

The JDBC type 3 driver, also known as the network-protocol driver is a database driver implementation which makes use of a middle-tier between the calling program and the database.

The middle-tier (application server) converts JDBC calls directly or indirectly into the vendor-specific database protocol.

This differs from the type 4 driver in that the protocol conversion logic resides not at the client, but in the middle-tier. However, like type 4 drivers, the type 3 driver is written entirely in Java.

#### **Type 4 Native-protocol (All Java)**

The JDBC type 4 driver, also known as the native-protocol driver is a database driver implementation that converts JDBC calls directly into the vendor-specific database protocol.

The type 4 driver is written completely in Java and is hence platform independent.

#### **Making a connection**

Next, the driver must connect to the DBMS: DriverManager is a class of *java.sql* package that controls a set of JDBC drivers. Each driver has to be registered with this class.

*getConnection(String url, String userName, String password):*

This method establishes a connection to specified database url. It takes three string types of arguments like

*url: - Database url where stored or created your database*

*userName: - User name of MySQL*

*password: -Password of MySQL*

```
Connection con = DriverManager.getConnection( "jdbc:db2:TEST", "db2admin", " db2admin " );
```

The object con gives us an open database connection

#### **5.7.2.2. Creating statements**

A Statement object is used to send SQL statements to the DB. It is an interface. Statement object executes the SQL statement and returns the result it produces.

*createStatement():*

It is a method of *Connection* interface, which returns Statement object. This method will compile again and again whenever the program runs. First we get a Statement object from our DB connection con

```
Statement statement = con.createStatement( );
```

Example

```
import java.sql.*;
```

```
class CreateProductTable
```

```

{
    public static void main(java.lang.String[ ] args)
    {
        try
        {
            Class.forName( "COM.ibm.db2.jdbc.app.DB2Driver" );
            String url = "jdbc:db2:TEST";
            Connection con = DriverManager.getConnection( url, "db2admin", "db2admin" );
            Statement statement = con.createStatement();
            String createProductTable = "CREATE TABLE PRODUCT " +
            "(NAME VARCHAR(64), " +
            "ID VARCHAR(32) NOT NULL, " +
            "PRICE FLOAT, " +
            "DESC VARCHAR(256), " +
            "PRIMARY KEY(ID))";
            statement.executeUpdate( createProductTable );
        } catch( Exception e ) { e.printStackTrace(); }
    }
}

```

Use the `executeUpdate()` method of the `Statement` object to execute DDL and SQL commands that update a table (INSERT, UPDATE, DELETE). We use the `executeQuery(...)` method of the `Statement` object to execute a SQL statement that returns a single `ResultSet`.

```

ResultSet rs = statement.executeQuery("SELECT NAME, PRICE FROM PRODUCT");

```

Typically, the SQL statement is a SQL SELECT `executeQuery(...)` always returns a `ResultSet`, never null. However, the `ResultSet` may be empty.

### 5.7.2.3. Process the result

#### ResultSet

`ResultSet` objects provide access to a table. Usually they provide access to the pseudo table that is the result of a SQL query. `ResultSet` objects maintain a cursor pointing to the current row of data this cursor initially points before the first row and is moved to the first row by the `next()` method.

```
ResultSet rs = statement.executeQuery("SELECT NAME, PRICE FROM PRODUCT");
```

Example

```
import java.sql.*;
class SelectProducts
{
public static void main(java.lang.String[ ] args)
{
try
{
Class.forName( "COM.ibm.db2.jdbc.app.DB2Driver" );
Connection con = DriverManager.getConnection( "jdbc:db2:TEST", "db2admin", "
db2admin " );
Statement statement = con.createStatement();
ResultSet rs = statement.executeQuery("SELECT NAME, PRICE FROM PRODUCT");
while ( rs.next( ) )
{
String name = rs.getString( "NAME" );
float price = rs.getFloat( "PRICE" );
System.out.println("Name: "+name+", price: "+price);
}
statement.close();
con.close();
} catch( Exception e ) { e.printStackTrace(); }
}
}
```

#### **5.7.2.4. Closing the connection**

It helps us to close the data source. The *Connection.isClosed()* method returns true only if the *Connection.close()* has been called. This method is used to close all the connection.

#### **5.8. Transactions commit and rollback**

Normally each SQL statement will be committed automatically when it has completed executing (auto commit is on). A group of statements can be committed together by turning auto commit

off, and explicitly committing the statements ourselves. This ensures that if any of the statements fail, they all fail. We can then roll back the transaction. When you do any operation such as , SELECT, INSERT, DELETE, or UPDATE in database then the transaction is committed after the execute update. This is default mode of transaction. To commit more than one transaction at a time you need to set connection object auto-commit(false) (conn.setAutoCommit(false);) and then execute the query and finally call conn.commit();. If any error occurs you can roll back your transaction by calling connection.rollback();

### 5.9. Accessing Metadata

JDBC has facilities to get information about a ResultSet or DB for a ResultSet, this information may include the number and names of the columns, the types of the columns etc. For a DB this information may include the name of the driver, the DB URL etc. This information about a ResultSet or DB is known as metadata

See the following classes for details:

ResultSet – see ResultSetMetadata

Database – see DatabaseMetadata

#### Getting metadata

Getting database metadata:

```
Connection con = DriverManager.getConnection( "jdbc:db2:TEST", "db2admin", "db2admin" );
```

```
DatabaseMetaData dmd = con.getMetaData( );
```

Getting ResultSet metadata:

```
Statement statement = con.createStatement();
```

```
ResultSet rs = statement.executeQuery("SELECT NAME, PRICE FROM PRODUCT");
```

```
ResultSetMetaData rsmd = rs.getMetaData( );
```

Example

```
import java.sql.*;

public class GetAllRows{

    public static void main(String[] args) {

        System.out.println("Getting All Rows from a table!");

        Connection con = null;

        String url = "jdbc:mysql://localhost:3306/";
```

```

String db = "jdbctutorial";
String driver = "com.mysql.jdbc.Driver";
String user = "root";
String pass = "root";
try{
    Class.forName(driver).newInstance();
    con = DriverManager.getConnection(url+db, user, pass);
    try{
        Statement st = con.createStatement();
        ResultSet res = st.executeQuery("SELECT * FROM employee6");
        System.out.println("Emp_code: " + "\t" + "Emp_name: ");
        while (res.next()) {
            int i = res.getInt("Emp_code");
            String s = res.getString("Emp_name");
            System.out.println(i + "\t\t" + s);
        }
        con.close();
    }
    catch (SQLException s){
        System.out.println("SQL code does not execute.");
    }
}
catch (Exception e){
    e.printStackTrace();
}
}

```