**SSD1: Introduction to Information Systems**

**SSD1: Introduction to Information Systems**

This is an introductory programming course. The course uses the Java language to illustrate programming concepts. You will learn how to program a computer to do simple tasks.

The course starts with HTML programming in unit 1. You will use the Web to search for information, and then you will design simple Web pages. You will also develop HTML forms that communicate with servlets.

In units 2 and 3, you will write simple Java programs that can be incorporated in your Web pages. In the process, you will learn and practice the following: compilation, syntax rules, variables, rudimentary object-oriented concepts (specifically classes, objects, and inheritance), data types, control structures, loops, and so on.

Each of the three units has several modules. Each module contains a homework exercise as well as both a multiple-choice quiz and a practical quiz. The course also contains three in-class exams. You can read about how to work through the quizzes, exercises, and exams in the course Help pages.

**SSD1 Hardware/Software Requirements**

* Internet connection
* iCarnegie-supported Web browser
* A text editor
* Sun's Java 2 Platform, Standard Edition (J2SE) 1.4 or later ([Appendix A. Java 2 Platform, Standard Edition (J2SE)](javascript:ContentByName('pg-sdk');))
* iCarnegie-supported MS Windows platform
* iCarnegie Servlet Workbench ( [Appendix B. The iCarnegie Servlet Workbench](javascript:ContentByName('pg-ic-servlet-workbench-appendix');) )
* WinZip application

**The purpose of SSD1 is for students to**

1. Learn foundational Internet skills and concepts
2. Learn foundational programming skills and concepts
3. Prepare for future Java courses by learning simple Java syntax and structure
4. Prepare for future programming courses by learning solid programming practices and strategies

**Students successfully completing SSD1 will be able to**

1. **Produce**
   * Basic HTML pages
   * Servlets that respond to user requests from browser
   * User interfaces with HTML FORM elements
   * Well structured solutions to simple programming problems
2. **Use**
   * The World Wide Web to find information
   * Effective Web searching techniques
   * A simple text editor to build and modify HTML and Java code
   * JDK tools for compiling and debugging Java classes and servlets
   * Simple Java data types, objects and collections
3. **Knowledgeably Discuss**
   * Basic HTML layout and publishing issues
   * The basic concepts of the software design cycle
   * The request-response architecture of clients, servers, and servlets in Internet applications
   * Rudimentary concepts of Object Oriented Programming
4. **Hold Positions in Quality Assuring and Troubleshooting Simple Web Sites**

On completing this course, successful students will be able to search the Web for information effectively. They will also be able to create simple Web pages using HTML and will be able to make servlets that respond to user input from browsers. Synthesizing these skills, successful students will be able to extend simple Web server functionality to create dynamic Web pages. For example, students will be able to write a simple calculator that performs arithmetic operations on numbers entered by a user.

The course also teaches effective strategies for learning programming languages, laying the foundation for students to learn advanced Java quickly, as well as other programming languages. This course will not only prepare students for future classes, but will also help them keep their programming skills up to date once they leave the classroom.

**Unit 1. The World Wide Web**

This unit of the course introduces you to the World Wide Web and its workings. You will learn how to use the Web as an effective tool for finding information. You will also learn how to create your own Web pages using Hyper Text Mark-up Language (HTML).

**1.1 Using the Web**

**1.1.1 Surfing the Web**

Let's start with some basic definitions. In case you do not already know, the Internet is a computer network that connects millions of computers across a number of countries. There is no central authority that controls the Internet; different organizations own different pieces of it. The Internet was originally conceived of by the Advanced Research Project Agency (ARPA) of the U.S. government in the 1960s. The **World Wide Web** (or **"the Web**" for short)refers to that portion of the computers on the Internet that can communicate with each other using a computer-network protocol called HTTP. All browsers use HTTP to request and receive Web pages from other computers.

Only a few years ago the expression "surfin' the Web" had meaning for only a small percentage of the population. Today that is no longer true, and if you haven't yet experienced surfing the Web, you should try it. Surfing (or browsing) the Web can take you to many different places with only the click of the mouse. However, if you do so, looking for information in a haphazard fashion, you *may* find the information quickly. Or, you may find yourself spending an entire evening wandering hither and thither, from page to page, finding at all sorts of interesting material—but not what you were actually looking for. So, in order to use the Web effectively, you need to know how to search the Web efficiently. That topic is covered in a later page.

As your search skills improve, you will find that you can locate a lot of information very quickly. But, be warned: all that means is that you are now able to search that many more places! Again, don't be surprised if entire evenings slip away while you meander around the Web. By the way, we have used the words ***page*** and ***place*** interchangeably when referring to the Web. Another synonymous term is ***site***, as in ***Web site***. All three terms refer to locations on the Web that you can visit and view through a browser. You are, in fact, familiar with at least one Web site: http://www.icarnegie.com/, the iCarnegie site you visited to log on to this course.

The Web is an interesting place. There is very little "law" on the Web—and it has been described as a place of "controlled anarchy." Efforts to regulate or govern it have been vigorously resisted by Web users. One way to look at the Web is to compare it to the early days of the Wild West, where just about anything could and did happen. While there are few rules, there are, however, two major areas of control. One has to do with the naming of a site on the Web and the other with the rules of an **Internet Service Provider** (**ISP**). (An ISP is any one of a number of companies that enable people like you and me not only to connect to the Internet and surf the Web but also to publish Web pages.) With respect to the first area of control, all Web sites must have unique names or addresses—one per machine. Therefore, if you want to set up a Web site, you have to apply to register a site name and pay a small fee; after that, no one else can use your site name. Concerning the second area of control, if you wish to publish Web pages, you must do so through an ISP, and ISPs often have rules that you must follow—rules that may govern, for example, the type of content Web pages may contain, where the pages can be stored, and so on.

Other than these constraints, the Web is a pretty wide-open place. However, on the Web you are still subject to societal rules and could find yourself in legal trouble if you slander someone, display information that is copyrighted by someone else, or commit fraud. When browsing the Web, you also need to understand that the people who use the Web and create Web pages represent a cross section of society. As such, there is both good and bad material on the web ("bad" material ranges from pornography to inaccurate or misleading information). And, because there is no one person or organization in control of the Web, there is no one to certify that the information presented on the Web is accurate, correct, and up-to-date.

Fraud is a problem, too. There have been several recent press releases from the federal government warning people to be cautious about buying unsolicited stocks and other securities over the Web.  Understand, however, that when fraud is detected, those responsible can be prosecuted, and the penalties can be severe. Nevertheless, when shopping on the Web, you would do well to remember the old saying, "a fool and his money are soon parted." This is not to say that there isn't a lot of good information on the Web and that you can't find good bargains; a surfer just has to be careful and evaluate any information taken from the Web.

One thing that the Web has facilitated is the formation of interest groups of various types. These groups can range from support groups for specific illnesses, to groups interested in hobbies such as stamp collecting or flying radio-controlled model planes. The Web provides a convenient meeting place for people with similar interests. Because distance is not a problem, people can easily communicate even though they are separated by vast distances. Some families that are geographically separated use the Web to communicate. People can see pictures of their grandchild almost as soon as they are taken. The parents have a digital camera (cost about $250) and they put the pictures on their Web page. The grandparents can access the Web page and see them.

If you have not yet tried searching the Web for information, you should do so. You might try looking for information on a favorite topic using a **search engine**. A search engine is a program that allows one to search for keywords in files at one or more Internet sites. Popular search engines include Lycos, Excite, and AltaVista. To try looking for information using a search engine, you might, first, pick a topic, then visit a search engine such as one of the ones mentioned above and read the help pages to learn how to use it. Then you are ready to start searching for information on your chosen topic. However, after every fifteen minutes or so of browsing, you should stop and see where you are. Don't be surprised if you find yourself far from the topic you chose originally, having become sidetracked by something else that caught your eye along the way. Also, be sure to remember that there is no one certifying that what you read is accurate. One important rule for Web surfers to keep in mind is "caveat emptor" or "buyer beware."

We discuss search engines in more detail in the course page [1.1.4 Searching the Web](javascript:ContentByName('pg-searching-the-web');).

**1.1.2 Your Web Pages**

As we have seen, there is no agency or group that certifies the information on the Web to be accurate, valid, and up-to-date. There is, however, the idea of the "Reasonable Person Principle" that many people follow. This idea is simply the Golden Rule at a practical level. If you are going to provide Web pages for others to view, then you should keep your Web pages in a condition that mirrors what you would like to find when you visit other people's pages.

At a minimum, the information should be correct and up-to-date. If your email address changes, then you should edit your Web page to reflect the change. If you don't want a lot of mail, don't include your email in the Web page. Putting your email address on the page implies that you will respond to mail. As a corollary to the "Reasonable Person Principle," if you have any doubt about displaying information, do not display it.

Remember that the users of the Web represent a cross section of society with its good and bad elements. Be careful about the personal information concerning yourself, your family, and others that you put on the Web. If you surf the Web, you will notice that many personal sites do not include home addresses and phone numbers. Many users keep the communication at an electronic level—probably not a bad idea.

If you decide to join the Web society, start small and simple. Keep your goals for your Web pages within your grasp. As your skills grow, you can enhance your Web pages to match your skills. If you build your pages in this manner, you will probably find your efforts more rewarding.

**1.1.3 Clients, Servers, and URLs**

**Clients**

**Client** and **server** are two terms that are common in computing contexts, but their meanings are not always understood. In a client-server setup, a client application requests information from a server application or asks the server to perform some task on its behalf. If you run a Web browser on your computer and use that browser to view Web pages from other computers, your browser is considered a client. When you, through a browser, request information from another site on the Web, the site that supplies the information (say, a Web page you wish to view) is considered a server.

**Servers**

Server applications are typically run on powerful computers, since they need to be able to service concurrent requests from a number of clients. On the other hand, client applications are typically run on less powerful computers, such as PCs or workstations. You probably do not use your home computer as a server, and if you publish pages on the Web, you probably do so through your school, your company, or through an independent ISP, which operates the server that makes your pages available to the Web.

**URLs**

The **Uniform Resource Locator** naming scheme provides users with a way to access Web resources using a uniform means for addressing those resources (HTML pages, text documents, images, and the like). A Uniform Resource Locator,or more commonly calleda "**URL**" (pronounced "U-R-L"), is the address of a specific Web resource, and typically, a URL has three elements. The first of these elements consists of an identifier that identifies the communication protocol to be used to access the resource the URL addresses, and this identifier is followed by a colon (:) and two slashes (//). Incidentally, a **communication protocol** is a generally agreed upon set of standards and rules that machines follow when they communicate with each other, and the most common protocol identifier found in URLs is "http" (which stands for **H**yper**t**ext **T**ransfer **P**rotocol); therefore, a large percentage of the URLs you'll encounter will begin with *http://*. Other common protocol identifiers you'll come across are "file" and "ftp"—as in *file://* and *ftp://*.

The second element of a URL is the name of the machine hosting the resource, an example being *www.icarnegie.com*. In addition, URLs typically include a third element, which is the name of the resource addressed by the URL. This name is specified in term of a path—such as, */courses.html*. Hence, a complete three-element URL might look as this: *http://www.icarnegie.com/courses.html*. And, as such, the URL might be interpreted this way: Using hypertext transfer protocol, retrieve the file on *www.icarnegie.com* named */courses.html*.

**How it Works — the Basic Model**

Client sends a request for a resource. A request consists of a protocol ("How do I get this resource?"), the server ("What server has this resource?") and the resource itself ("What resource specifically?"). The server first locates the resource that has been requested. Using the specified protocol, it then transmits a copy of the bytes that constitute the requested resource back to the client. When the client receives the resource, it deals with it in an appropriate way (saves it, displays it, etc.).

For a more specific discussion of this topic with regard to Web servers, read the [Webopedia](http://webopedia.com" \t "externalWindow) article "[How Web Servers Work](http://www.webopedia.com/DidYouKnow/Internet/2003/HowWebServersWork.asp" \t "externalWindow)."

**Beyond the Basic Model**

Under-utilized computer resources on both the client and server sides of things, the need for dynamic Web content, as well as a number of other issues—security being the most important of these—have encouraged efforts to extend the basic client-server model. As it turns out, these developments have focused on what occurs on the client and server ends, after a transmission from one to the other has taken place. To begin with, these innovations consisted of client-side and server-side scripting. On the client side, this means that some HTML pages now execute scripts. On the server side, this means that servers now sometimes execute auxiliary applications, including Java servlets. The importance of Java servlets lies, in particular, in their ability to address the need for dynamic content by composing Web pages "on the fly." This aspect will be the course's focus in later units.

**Questions to Consider**

* Given the description of Web servers above, what limitations can you see with this client-server model?
* In the above model, can the client receive personalized information?
* In your own explorations of the World Wide Web, what behaviors have you seen that the model above does not account for?

**1.1.4 Searching the Web**

First, let's discuss the difference between ***browsing*** and ***searching***. There is no formal definition for these terms like the kind we might find in a science textbook. However, for our purposes here *browsing* denotes the activity of following hyperlinks (also called *links*) that one encounters on succeeding pages. For example, suppose you hear a URL in a radio commercial on your way to school, and when you get a chance, you sit down and access that site. At that site, you find many hyperlinks, which in most browsers are displayed as blue underlined text. Among the many hyperlinks, you find one that catches your interest, and you click it. The link takes you to another page, which in turn contains another interesting link that you proceed to click. And, so it goes, until the next thing you know it's past lunchtime, and you've missed two classes and your work-study job.  *Browsing* denotes this unfocused, random passing from link to link around the Web.

Although what begins as *searching* may turn into browsing, searching involves the use of a computer program called a **search engine**. We said a little about search engines earlier in the course, and now it's time to say a little more. The term *search engine* is a generic one that refers to a class of programs, though the members of this class may employ very different search mechanisms. On the Web, popular search engines are often available through particular Web sites, sites that feature an index of almost all the pages on the Web. Users can therefore use the search engines at any of these sites, which rely on the mechanism of the site index, to search the Web efficiently for documents and pages containing specific words or phrases. The results of such a search is a list of Web pages the engine has found (referred to as "hits"), arranged in order of decreasing relevance. Several engines of this type are available on the Web, some of which are listed below:

* [Excite: www.excite.com](http://www.excite.com" \t "externalWindow)
* [AltaVista: www.altavista.com](http://www.altavista.com" \t "externalWindow)
* [Lycos: www.lycos.com](http://www.lycos.com" \t "externalWindow)

Note also that there are two types of search sites on the Web:

* **Sites that feature a search engine**. Sites of this type maintain indexes or databases of the addresses of virtually all of the pages and documents on the Web. These indexes and databases are updated regularly and automatically by programs known as "spiders," which search the Internet for the addresses of new documents and pages that have become available. Therefore, when you use the search engine at any of these sites, you are, practically speaking, searching almost the entire Web.
* **Sites that feature a Web directory**. A Web directory is similar to a telephone directory, in that it organizes information available on the Web into different categories and subcategories. Typically the categorization process relies on some amount of human effort, and since the amount of information on the Web is vast, more than is possible for one company or person to categorize, only a fraction of what is on the Internet gets included in a Web directory. Unlike search engine sites, which rely on automatic means to gather entries, Web directory sites typically rely on the owners of relevant Web sites to submit their sites for inclusion. One of the better-known Web directory sites is [Yahoo!](http://www.yahoo.com" \t "externalWindow); however, nowadays almost all search engine sites also have directories.

It turns out that, if you are looking for specific information and want to find it in a reasonable amount of time, you have to refine your searching skills. To do this takes practice and information. If you look carefully at a search engine's Web page, you will find a help button somewhere. Clicking this button will take you to a set of help pages. It will take time to work through this information and to test it, but it's time well spent, because it will reduce the overall time you spend finding information.

Let's try a sample search. We would like to learn something about tourist activities on the island of Java in Indonesia. Say we go to [Altavista (http://www.altavista.com)](http://www.altavista.com" \t "externalWindow) and type in the word "java" into its search engine. AltaVista pauses for a second and then comes back with a lot of "hits." You can see the actual output we got when we tried this: [See Search Output](http://www.icarnegie.com/content/SSD/SSD1/2.1/normal/pg-www/pg-using-web/pg-searching-the-web/java1.gif" \t "externalWindow). As you no doubt noticed looking at the output, the results we got were not what we wanted. Many of the pages returned by the engine were about Java the programming language, not Java the island. Although search engines these days are quite clever, they cannot read our minds and give us exactly what we want all of the time. So, we must refine our searches. Look at AltaVista's help page for the different operators available to us. Since we know that the island of Java is in Indonesia, we can use the "+" operator to specify that the word Indonesia must appear in all result pages. We are thus reducing the scope of our search for Java to those pages that contain references to Indonesia as well. The pages returned by our refined search will, we hope, give us the information we desire. When we tried our refined search, that hope was realized—as the [results of our revised search](http://www.icarnegie.com/content/SSD/SSD1/2.1/normal/pg-www/pg-using-web/pg-searching-the-web/java2.gif" \t "_new) show.

As a prudent use of the "+" operator in the search string shows, a little knowledge can go a long way in helping you search for information. You don't need a book if you are willing to use the help information offered by each search engine. It is important to note here that each search engine has different rules for refining and narrowing a search. You can learn more by visiting the search engine's help pages. We urge you to spend time using a variety of search engines, trying to refine and narrow the focus of your searches. Doing so is fun, and what you learn can be helpful when it comes time that you really need to find information.

**1.1.5 Commerce on the Web**

Web-based Internet commerce (also known as e-commerce) can take many forms, of which some are relatively new. We will look briefly at three of the most common forms of Internet commerce:

* Internet Service Providers (ISPs)
* Advertising
* Commercial Transactions

**Internet Service Providers**, as the name implies, sell you access to the Internet. The cheapest form of internet service provided is the ability to log on from your home. You typically dial up to a number provided by the ISP and are connected to the Internet, after which you can browse and search the Web. Many ISPs also provide email service, and many allow some minimal Web publishing. However, although one still needs an ISP to establish a major Web presence, establishing and maintaining such a presence can be expensive. The price is usually based on the total size of your Web page files and how many hits you want your site to be able to handle in a given amount of time. If you are going to sell goods through your Web site and are looking to make it popular, it will have to be capable of handling a large number of hits, and that can be expensive. Until recently, ISPs gave you a maximum number of connect hours to the Web for a fixed fee.  If you exceeded that number of hours, you incurred an hourly charge. However, things have changed, and many ISPs now offer unlimited Web connect time for a fixed fee—currently about $20.00 a month.

**Advertising** is one way that search engine companies make money. If you took our advice in [1.1.4 Searching the Web](javascript:ContentByName('pg-searching-the-web');), you tried out one or two search engines, searching for variety of things. While you were doing this, you probably noticed lots of visual clutter that search engine companies call advertising. The companies that this "visual clutter" advertises pay a search engine company a fee every time their ads are displayed. The fee per display is small, but since there are a lot of people using the Web, the display fees add up. Search engines also select the ads to be displayed on the basis on what a user is searching for. If, for example, a user is looking for information about football, they probably will not be shown ads for the opera. However, if they are searching for classical music, a search engine may show ads to the opera. Note also that many of the ads include links to the Web site of the company they advertise. Follow just a few of these links and you may find yourself spending a lot of time and, possibly, a lot of money.

**Commercial transactions** do not represent a large percent of Internet commerce, but that percentage is growing. Presently, you can order books from several companies on one day and receive them the next. Furthermore, during the holiday season, you can also order toys, and companies that use printed catalogs are now likely to put their catalogs on the Web: it is cheaper to display a catalog on Web than to print and mail it to many individuals. Doing this also makes inventory control a little easier; the Web-based catalog can be updated immediately to reflect changes in inventory. Therefore, if blue shirts sell out, that listing can simply disappear from the catalog on the Web. That is not possible with a printed catalog, and some electronic catalogs display more items than it is practical for a printed catalog to do. To see an example of a sophisticated commercial site that features a Web-based catalog, check out the Lands' End Web site at [www.landsend.com](http://www.landsend.com" \t "externalWindow).

Recent news articles report that many observers believe that the number of commercial transactions on the Web will continue to grow each year for the foreseeable future, and other forms of Web commerce are possible. Web commerce is, in fact, a field fertile with opportunity for clever and thoughtful individuals. There are many niches that haven't even been filled yet, and Web commerce as an enterprise is still very new. Just remember that prior to 1997, you couldn't even order books on the Web.

**1.1.6 Some Ethical Considerations**

Let's start with some interpretations of the simple question, "What can you put on your Web page?"

1. Can you go to your favorite sports team's Web page, make a copy of the .jpeg or .gif file that is their logo and put that logo on your Web page?
2. Can you put a link on your page to the home page of your favorite sports team?

The answer to number 1 is "No." There is a very strong possibility that your favorite sports team has that logo copyrighted. To use it legally, you need their permission. Whether or not they prosecute you for copyright infringement is beside the point: you need their permission to put the logo on your page.

The answer to number 2 is "Yes." You can put on your Web page a link to any other Web page. If your favorite sports team does not want everyone to access its Web page, it can put a gate on the page and require people to log in with a password.

The use of any material from another Web site that is not specifically marked as free for the taking, or for which you do not have permission, is not ethical. In this context, the word *material* is very broad. It means text, pictures, graphics, backgrounds, cute little buttons, and enhancements. There are some sites that provide graphics and enhancements free for the taking. If you find something you like, you can send email to the Web page owner and ask for permission to use what's there.

In the page [1.1.2 Your Web Pages](javascript:ContentByName('pg-your-web-pages');), we said that if you have any doubt about material you want to put on your Web page, leave the material out. The same is true here—if you don't own it, don't use it. And, if you use text, be sure to include a full attribution stating its source.

**1.2 What's in the Web?**

**1.2.1 What's in a Web Page?**

By now, you probably have a basic knowledge of Web browsers and how to use them. You may also be aware that there are several competing browsers on the market. Web browsers allow us to view Web pages, which is, as you know, what you are currently reading. A Web page is "written" in a formatting language called HTML. When we view a Web page, we normally do not see the elements of this language. However, if you are using a browser such as Netscape or Internet Explorer, it is possible to see the HTML elements that make up the page you are viewing (known as the page's "source code" or just "source" for short). Position the mouse pointer anywhere over the page and click the right mouse button. Click **View Source**, **View Frame Source**, **View Page Source**, or equivalent command on the menu that appears. However, be warned: some pages are quite complex, so we suggest you start with a simple one. To see one example, you might look at the [author's home page](http://www.cs.cmu.edu/%7Ejar" \t "externalWindow). It is a relatively simple Web page, but it manages to provide some useful information. Take a moment to look at the page, and then look at its source.

The type of file you are looking at when you view a page's source is a **text** **file**. Unlike Claris, WordPerfect, or Microsoft Word files, text files do not contain special formatting characters that cause text to appear as **bold**, underlined, and the like. But, if the file that contains a Web page's source contains no special formatting characters, how is that Web pages exhibit these and many other formats? The answer is HTML consists of special formatting notation—called *tags*—that tell a browser how to display a Web page's content. Some of these tags can create connections to other computers over which information is transferred (such as text and images) for display on your computer. Now, to look at the tags that create this course page, position the mouse pointer anywhere over the page and click the right mouse button. Click **View Source**, **View Frame Source**, **View Page Source**, or equivalent command on the menu that appears. In a short while, you are going to learn a number of these tags and use them to create your own Web page.

Since you are learning to create Web pages, it is a good idea to get in the habit of looking at the Web page source when you browse. Don't be surprised if you find some very strange information in the source files. The more detailed and complex a page is, the more complicated its source is, in terms of number and types of tags. But, any page's source is understandable—if you take the time to decipher the tags. However, since we have many other things to cover in this course, we will decipher only a few tags for now—just enough to allow you to produce several basic Web pages. If you want to take your Web pages to the next level of visual delight, then you will have to explore the tags needed to do that on your own (and we strongly encourage you to do just that kind of exploration).

We should actually call the programming exercises in this course "labs"—since in order to get the most out of them, you should always do a little experimenting when you are working on them. Of course, unlike what you may remember from Chemistry class, these "labs" should allow you to experiment freely as you like—without the need to wear safety goggles and aprons. Such experimentation is one of the major ways we learn how to write Web pages or Java programs—by mimicking those features, we see and like in the applications of others—and by trying out features, we dream up on our own. Programming is like playing an instrument; it takes practice, frequent practice on a regular basis, and many of our students find experimental practice one of the more enjoyable aspects of programming. Of course, with experimentation comes failure—but, again, one of the biggest advantages programming has over Chemistry is that simple programming experiments do not result in explosions, flames, and broken glass!

**Other Information on Creating Web Pages**

There are a lot of books out about authoring Web pages. One that the author has both used and loaned to others is *HTML Web Publishing 6-in-1*, by Todd Stauffer, from QUE publishers (ISBN 0-7897-1407-8).

You can also visit the sites behind the links below and find a lot of good information about HTML:

* Raggett's [Getting Started with HTML](http://www.w3.org/MarkUp/Guide/" \t "externalWindow)
* [A Beginner's Guide to HTML](http://www.ncsa.uiuc.edu/General/Internet/WWW/HTMLPrimerAll.html" \t "externalWindow)

Also, before you spend your money on any book, we suggest you check your campus library to see what they have. And, you can always "surf the net" to find online tutorials as well.

**1.2.2 A First Look at HTML**

The acronym ***HTML*** stands for ***Hypertext Markup Language***.  Let's examine this term, starting with the last word first:

* ***Language***. Language is a term we all understand. A language is a system of signs used for communication—written and oral. For example, we are communicating the content of this course with a language known as *English*.  But, English is a human language, a language that humans use for communicating with humans; HTML, on the other hand, is a language that humans use for communicating with Web browsers.
* ***Markup***. In paper publishing contexts, markup is the process of preparing manuscripts for typesetting by marking them with directions about font type and size, indentation, spacing, and the like. The same principle applies in electronic publishing. Instead of marks made by pencil or pen, formatting directions are conveyed by notation called "tags." We mentioned tags in this context earlier; however, we did not mention earlier that the goal of markup is to make the text more meaningful. Whether markup elements make one part of a text bigger, darker, or a different color than another part, or whether they add blank lines, indentation, or horizontal bars, markup elements should make the text more meaningful by focusing and guiding a reader's attention. For example, a reader is more apt to notice and recognize a title for what it is if it is bigger and darker than the rest of the text.
* ***Text***. Text refers to the words on the computer screen that we are marking up. But, the word "text" actually has another, special meaning in computer contexts: the term "text format" denotes the plain characters of the ASCII (American Standard Code for Information Interchange) character set. As it is, text format is the simplest format in which to store written materials, and text files contain only the characters of the ASCII character set— which consists of printable characters such as punctuation marks and alphanumeric characters, but not symbols that render text in special formats such as **bold**, underscore, and *italics*. In contrast, word processors like Microsoft Word embed all sorts of hidden characters in their files, and these hidden formatting characters control the appearance of text as we see it on the screen.
* ***Hyper***. This term should be combined with the term "text" in this discussion. Hypertext as a concept developed from Vannevar Bush's 1932 visionary brainchild known as the "memex," a device that would create links between related topics in different papers, and although the memex was never built, the notion of creating linked, nonlinear structures of information endured. A hypertext document is quite different from a traditional printed document. In a printed document, the order in which information is presented is linear—that is, sentence 1 precedes sentence 2, which in turn precedes sentence 3, and so it goes on up the levels so that chapter 1 precedes chapter 2, which precedes chapter 3, and so on. However, a hypertext document contains links, and by clicking a link, a user is taken to a different page or different part of the same page. If the user then clicks other links, he or she is taken to another page or part of the same page. Thus, the user can follow a nonlinear path through the document or documents, using the links provided by the author and encountering the information there in an order that depends on the choices that both they and the writer have made.

**1.2.3 A More Detailed Look at HTML**

The remarkable fact about HTML is that it has already become a widely accepted standard, and browsers that can display HTML documents have become commonplace and available for most types of computers. Therefore, most computers, given an appropriate browser, can display HTML files. This was not the case in the days before HTML. Back in those day there were several competing formats, each requiring different software to read and display its documents—and getting the software to display a particular format on a particular type of machine was often not an easy task.

**Choice of Editor**

In this subsection, we will practice writing HTML documents and viewing those documents in your browser. In order for you to write an HTML document in a computer, you need an editor. The editor that you choose must allow you save documents as text files.  The editor must also allow you to save these documents with  ***.html*** extensions (**.*htm*** will work also): "MyFirstWebPage.html" or "MyFirstWebPage.htm".  A **filename** **extension**, as you may know, is a suffix on a filename that consists of a period (or "dot") and a sequence of characters. By saving files in text format, you force the word processor to leave out its own special formatting characters, which Web browsers cannot read.  By using the extension .html, you identify the file as one a Web browser can read and display using the HTML markup tags in the file. In Windows, the extension tells the operating system the kind of contents stored in the file—that is, the file type. For example, an .htm extension indicates an HTML file type. On UNIX and Macintosh, the filename extension does not mean anything to the operating system.

You really must do a lot of practicing in addition to doing the exercises. The exercises will be much easier and more worthwhile if you have spent several hours just experimenting with HTML. Remember that programming is like playing an instrument—you must practice. Practicing involves spending time every day on the computer experimenting and trying things out to see what happens. Practicing one hour a day, seven days a week is much better than practicing for three on Saturday and four on Sunday.

**The Basics**

As we mentioned earlier, the symbols that we add to the text to tell Web browsers how to display that text are called *tags*, and there are many types of tags. Each tag is enclosed inside a pair of angle brackets ( "<" and ">", otherwise known as the mathematical operators for "less than" and "greater than"). Some tags come in beginning and ending pairs that follow the syntax <TAGNAME> and </TAGNAME>, with ending tags distinguished by a forward slash ("/") after the open angle bracket ("<"). Here is a pair of tags that identify a section of text as a paragraph:

<P>   
  
 Now is the time for all good programmers to quit.  
  
 </P>

The *<P>* tag above marks the beginning of a new paragraph, and the *</P>* tag marks the end of the paragraph. You can use either an uppercase or a lowercase *P,* but our convention is uppercase. The text between these tags will be displayed so that they "wrap" to fit in the window of the browser—that is, lines of text will not extend beyond the window's left or right margins.

But, not all tags come in pairs of beginning and ending tags. Suppose you want the Web browser to put a line break at a particular place in the text. Just hitting the ENTER key (or the RETURN key on a Macintosh keyboard) while editing a file will cause the editor to insert a line break in the text—but doings so will not cause a Web browser to display a line break at that point. That is because the character that hitting an ENTER key inserts into a text does not mean the same thing to a browser that it does to an editor. To tell a browser to display a line break, you use the <BR> tag. Note that while you can use several consecutive <BR> tags to tell browsers to display multiple blank lines (that is, a series of line breaks), this will not always have the desired effect: some browsers will display multiple blank lines, while others will ignore all <BR> tags after the first. Here is an example of a passage of text inside paragraph tag that makes use line break tags:

<P>This is the first sentence of the paragraph   
  
and the browser will add line breaks wherever  
  
needed to fit it on the page.<BR>This will be  
  
a new line in the display.<BR>So will this one.</P>

We can make the text easier to read by typing the above text like this:

<P>  
  
This is the first sentence of the paragraph   
  
and the browser will add line breaks wherever   
  
needed to fit it on the page.  
  
<BR>  
  
This will be a new line in the display.  
  
<BR>  
  
So will this one.  
  
</P>

In either case, a browser will display something like the following:

This is the first sentence of the paragraph and the browser will add line breaks wherever needed to fit it on the page.   
This will be a new line in the display.   
So will this one.

However, formatting the text as in the second case can makes the text easier to work with in an editor.

Note: the browser will ignore all of the extra white space we've added to make the text easier to read and will simply display the text according to the instructions provided by the HTML tags in the text. But, we are getting ahead of ourselves.

**The Main Components**

An HTML document should begin and end with this tag pair:

<HTML>

</HTML>

Immediately after the <HTML> tag should be the following tag pair to denote the document header:

<HEAD>

</HEAD>

In between the <HEAD> </HEAD> pair should be the <TITLE> </TITLE> tags. The title of the document is specified within them. This title will be displayed in the title bar of the Web page—for example,

<HTML>  
  
<HEAD>  
  
<TITLE>  
  
My First Web Page  
  
</TITLE>  
  
</HEAD>  
  
</HTML>

Next come the beginning and ending BODY tags:

<HTML>  
  
<HEAD>  
  
<TITLE>  
  
My First Web Page  
  
</TITLE>  
  
</HEAD >  
  
<BODY>  
  
</BODY>  
  
</HTML>

Almost all of a Web page's content will come between the beginning and ending BODY tags ("<BODY>" and "</BODY>").

Now, we need to consider one other type of tag—"comment" tags. If, for example, we are creating a Web page and we want to put something (a remark, note, etc.) in among the page content but we DO NOT want that something to be displayed by a browser, we can accomplish this by putting inside a pair of comment tags. Unlike the label-like tags we have encountered so far, comment tags consist of a beginning and ending sequence of characters ("<!--" and "-->"). Below is an example of a comment inside a pair of comment tags:

<!-- This is a comment. -->

The <!--   --> character sequences above mark or *delimit* (a good programming word) the beginning and end of the comment.

**Physical Styles**

**Physical style** tags allow us to tell browsers to use certain font styles. The following tag pairs force the following styles:

* <B> </B>                 **Bold**
* <I> </I>                    *Italics*
* <TT> </TT>             Monospace(fixed width)
* <U> </U>                 Underline
* <SUB> </SUB>       Subscript
* <SUP> </SUP>        Superscript

If an HTML document had the following text:

The following is <B>bold</B> while the following is <U>underlined</U>.

It would display like this:

The following is **bold** while the following is underlined.

**Logical Styles**

When we use **logical styles**, we allow browsers to decide how to display the text. Here are a few of the logical style tags:

* <EM> </EM>                                 *Emphasized text*
* <STRONG> </STRONG>             **Strongly emphasized text**
* <CITE> </CITE>                           *Text in citation*
* <SAMP> </SAMP>                       Text in a computer screen output sample

There are a few other logical style tags, so you might want to go exploring here.

These are used in a manner similar to the way physical tags are used, but the browser decides how to display the text. For example, the following HTML text:

The following is <STRONG> strongly emphasized text </STRONG>.

might look like this in Netscape

The following is **strongly emphasized text**.

and like this in Internet Explorer

The following is strongly emphasized text.

**Size and Color**

If you want to set the background color of your Web page, you can do this in the body tag like this:

<HTML>  
  
<HEAD>  
  
<TITLE>  
  
My First Web Page  
  
</TITLE>  
  
</HEAD >  
  
<BODY  BGCOLOR=RED>

</BODY>  
  
</HTML>

You can set the font size as follows: <FONT  SIZE=7>

Possible font sizes range from 1 (smallest) to 7 (largest).

You can also set the color of the font as follows: <FONT  COLOR=RED>

The color of the font following the above tag will be red until it is changed by another FONT tag—or until the next ending FONT tag ("</FONT>"), at which point the font will return to the default color, whatever that happens to be set to.

Choosing colors by name limits you to a basic set of colors, and if you want to work with shades of color, you will have to employ RGB color codes. A RGB color code consists of a sequence of three two-digit hexadecimal values, values that can range from 00 to FF  The RGB color system uses the three hexadecimal values to represent the amounts of red, green, and blue (in that order) present in the desired color. The amounts specified can range from 00 (none) to FF (as much of the color red, green, or blue as possible). Thus, to express a color, you employ a six-digit hexadecimal number representing three color-amount values. Each color amount value is represented by two hexadecimal digits.  For example, instead using the notation "<BODY  BGCOLOR=RED>" to set the background color, as we did above, we can achieve the same result using an RGB color code:

<BODY  BGCOLOR="#FF0000">

To specify yellow as the background color, one could write

<BODY  BGCOLOR="#FFFF00">

To specify gray, one could write

<BODY  BGCOLOR="#BEBEBE">

**Headings**

There are seven levels of heading tags, which come in beginning and ending pairs. The <H1> and </H1> tags are used to delimit the highest-level headings (which are also the largest) and <H7> and </H7>, to delimit the lowest level headings (also the smallest).

**Horizontal Lines**

It is possible to create a horizontal line (or "rules") on a Web page by using an <HR> tag. Used without attributes, an <HR> tag creates a shaded line across the width of a page—and should the user change in the size of the window in which the Web page is displayed, the browser will adjust the length of the line accordingly. However, there are several attributes that allow you to modify the appearance of the line the tag creates:

|  |  |
| --- | --- |
| SIZE=5 | SIZE sets the line's thickness in terms of pixels—5 pixels in this example |
| WIDTH=50 | WIDTH sets the line width in terms of a percent of the width of the Web page displayed—50% in this example |
| ALIGN=LEFT | ALIGN justifies the line against either the right or the left margin (as in this example) or centers it in the window |
| ALIGN=RIGHT | justifies the line against the right margin |
| ALIGN=CENTER | centers the line in the window |
| NOSHADE | NOSHADE removes the line's default shading |

And, you can use several of these attributes together like so:

<HR SIZE=10 WIDTH=75 ALIGN=CENTER>

**Preformatted Text**

Suppose you have text that is already formatted the way you want it, and you don't want to trust browsers to decide how it should be displayed. You can make browsers use the formatting you designate by using the PRE tag (where *pre* stands for *preformatted*), which consists of the beginning and ending tags <PRE> and </PRE>. Here is an example:

<PRE>  
Type this #$&\*\*$&($\*\*  
 |\_\_\_\_\_\_\_\_\_|  
Be sure to type exactly this.  
</PRE>

Ideally, a browser will display text just as you have formatted it. However, browsers may not always present text as intended. Therefore, you should always verify that the page displays in the browser as you intended it to.

**Lists**

Here we encounter two kinds of lists: ordered (or numbered) lists and unordered lists. Creating an ordered list requires two different tags. The first is the OL tag, which consists of the beginning and ending tags <OL> and </OL>. The OL tag automatically numbers the elements of a list, starting at 1. The second tag required to create an ordered list is the LI tag (for "list item"). Use the tag to create items in the order list by placing a single <LI> tag in front of each element of the list. Here's an example:

The Top Three Reasons to Write Java Programs:  
<OL>  
<LI> I want to pass   
<LI> I want to graduate  
<LI> I want a job when I graduate  
</OL>

An unordered list, on the other hand, requires use of the UL tag, which also consists of a pair of beginning and ending tags: <UL> and </UL>. Like the ordered list, however, items are created on an unordered list using the LI tag:

The Top Three Reasons to Write Java Programs:  
<UL>  
<LI> I want to pass  
<LI> I want to graduate  
<LI> I want a job when I graduate  
</UL>

Finally, list items of ordered and unordered lists may optionally be enclosed in a beginning-ending pair of list item tags:

The Top Three Reasons to Write Java Programs:  
<OL>  
<LI> I want to pass </LI>  
<LI> I want to graduate </LI>  
<LI> I want a job when I graduate </LI>  
</OL>

and

The Top Three Reasons to Write Java Programs:  
<UL>  
<LI> I want to pass </LI>  
<LI> I want to graduate </LI>  
<LI> I want a job when I graduate </LI>  
</UL>

**Finally**

This has been a very brief look at some of HTML's capabilities. There are many more things you can accomplish using HTML tags; however, with the tags you've encounter here, you have all you need to start building good, effective Web pages. We encourage you to, first, work with this material found in this page and then to explore some of HTML's other capabilities. One word of caution: HTML is evolving rapidly, and new capabilities are constantly being added to the language; therefore, if you want to master HTML and maintain that mastery, you will have to be diligent to keep up with it as the language keeps changing.

In learning HTML, just as in any other learning, whether it's learning to play tennis or to play piano, remember to practice one hour a day. Who knows, you might even enjoy it!

**1.2.4 Creating Links to Other Documents**

It is worth recalling the previous discussion of HTML documents. We mentioned that HTML documents can have links to other documents and that users can use these links to navigate to these other documents. Note that in the context of the Web, all documents and other Web resources reside on some computer. In this section, we will return to the subject of how Web resources, such as documents, can be distinctly addressed so that links can be created to those documents from other documents. As we learned earlier, such distinct Web addresses are called Uniform Resource Locators or URLs. In this section, we will return to this subject and go into it in a little more depth.

**URLs**

To review what we learned earlier, URLs are the addresses of Web resources that allow us to access those resources. A typical URL consists of three elements, though this number can and does vary. These elements are

1. A **protocol identifier,** which consists of a label signifying a communication protocol (such as *http*, *file*, and *ftp*) followed by a colon (*:*) and two slashes (*//*), as in http://
2. A **host name** (or "server name" as it is often referred to), which is the name of the machine hosting the resource, as in www.icarnegie.com
3. A **resource name** in terms of a *path*, as in /courses.html

Now, we'll explore these in more depth. Here is the complete URL to the author's home page:

http://www.cs.cmu.edu/afs/cs/user/jar/www/index.html

Let's take this path apart piece by piece.

The first element, *http://,* indicates that the home page is to be retrieved using the communication protocol known as HTTP (Hypertext Transfer Protocol).The second element, *www.cs.cmu.edu*, is the symbolic name for a computer at CMU (Carnegie Mellon University). That computer is the Computer Science Department's Web server and is the computer that hosts the author's home page. Incidentally, by convention, host names commonly begin with the node *"www*," although they can just as easily begin with something else—such as "*server*," as in "*server.cs.cmu.edu*." The last node of this symbolic name, "edu," adheres to the Domain Name System (DNS) convention of identifying educational entities with the suffix *"edu*."  Below are a few other DNS conventions, showing suffixes and the entity types those suffixes go with:

* com – commercial organizations such as Ford or IBM
* edu – U.S. educational entities
* gov – U.S. government agencies
* net – a network service provider
* org – non-profit organizations such as PBS or the Red Cross

After specifying the host computer name, we then provide more information about the exact location of the home page file on the host in terms of a "path"—that is, in terms of a meaningful series of characters and slashes (/). The file name of the home page we are trying to access is the string of characters that follows the last slash on the right in the path. The slashes and characters to the left of the file name describe the directory hierarchy, from the server root directory—signified by the initial slash (/) following the server name—to the directory in which the file resides. The *directory system* (or the *folder system* on machines running Mac OS or Windows 9x) is hierarchical in a way that is similar to a family tree. In our example, the initial slash (/) after the server name signifies the root of the directory tree described by the URL http://*www.cs.cmu.edu/afs/cs/user/jar/www/index.html*. (Incidentally, root directories are sometimes represented by appending the server name to the slash, like this *www.cs.cmu.edu*/.) Moreover, each slash (/) that follows the slash after the host name marks a transition to a lower directory level—much the way that branches of a family tree lead from grandparent to parent to child to grandchild. Let's take a closer look at the part of the URL that follows the host name, which we have called a "path":

/afs/cs/user/jar/www/index.html

The term "path" provides us with a metaphor by which we might understand this segment of the URL. This segment indicates that in order to access the file *index.html*, we must first enter the *afs* subdirectory of the server, and then once in the *afs* subdirectory, we must go to the *cs* subdirectory. We then must continue in a similar manner and enter the *user* subdirectory, the *jar* subdirectory (which is the author's personal directory), and, finally, the *www* subdirectory. Remember, in a path, all of these subdirectory names are separated by slashes (/).

To reinforce our understanding of the hierarchical nature of URLs, say we wanted to create a URL to Jim's physical office space at CMU. That URL might look something like the following:

wean.cmu.pgh.pa.usa/floor-3/corridor-1/room-08.

**URLs and Links**

So far, in our discussion of URLs, we have presented them from the perspective of complete and unique addresses necessary for identifying and accessing resources on the Web. A critical element in our discussion so far has been this notion of *complete*: in order to access a Web resource, any resource, a client such as a browser must use a complete URL. However, as we move to our discussion of links—those elements in Web pages that allow users to navigate from one resource to another—it is important to make the point that, although browsers require complete URLs, links do not need to *supply* complete URLs. That is, a link can supply only part of a URL (what we'll call a *relative URL*), and the browser relies on other sources to derive the complete URL.

Now, note that we are talking about links and linking here only at a conceptual level and haven't said anything yet about how links are implemented with specific HTML elements. However, don't worry: we will get to that matter shortly, later in this page.

**Links and Relative URLs**

*Relative URLs* are URLs that, at the upper end of the range, lack only a protocol identifier and a host name. At the lower end of the range, relative URLs can consist of only a resource file name. That is, the most a relative URL can contain and still qualify as a relative URL is what we have come to know as the *resource name* (as in */afs/cs/user/jar/www/index.html*), and the very least a relative URL can contain and still qualify as a relative URL is the *resource file name* (as in *index.html*).

There are three other points to make here.

* A relative URL that begins with a slash (/) describes the location of a resource in relation to a host's root directory. Therefore, we might call that type of URL a *relative-to-root URL*.
* A relative URL that does **not** begin with a slash (/) describes the location of a resource in relation to the location of the resource (that is, page) containing the link. Therefore, we might call this type of URL a *relative-to-resource URL*.
* The value of relative URLs is that they, for one, give us a way of abbreviating URLs, thus making it easier for us to create links. There is another benefit with respect to link maintenance, and we'll go into that shortly.

A relative-to-resource URL is a good choice, if you want to create a link to a file that is in the same directory as the file containing the link, or is in a directory that is a subdirectory of that directory, or is in some other directory on the same computer. If the file is in the same directory as the file containing the link, the relative URL need only include the minimum amount information: the name of the file to which the link is being made. If the file is in a directory that is below the directory as the file containing the link is in, the relative URL need also include, in addition to the name of the file, the path of the additional directories to where that file resides.

However, remember that a browser needs a complete URL to access a resource. Therefore, when it encounters a relative URL of either type, it must derive a complete URL by obtaining the missing information from some other source. However, while there are two or three sources a browser might obtain this information from, for our purposes in this course, the only source we'll mention is that of the URL of the page containing the link.

That said, when a browser encounters a relative-to-resource URL (the type that does **not** begin with a slash), it constructs a complete URL by prepending the URL of the file containing the link—less the file name—to the relative-to-resource URL.

For example, if the page containing the link has this URL:

http://*www.cs.cmu.edu/afs/cs/user/jar/www/index.html*

and the relative-to-resource URL consists of

*courses.html*

the browser would construct the following complete URL:

http://*www.cs.cmu.edu/afs/cs/user/jar/www/courses.html*

Again, if the page containing the link has this URL:

http://*www.cs.cmu.edu/afs/cs/user/jar/www/index.html*

and the relative URL consists of

*fall2003/courses.html*

the browser would construct the following complete URL:

http://*www.cs.cmu.edu/afs/cs/user/jar/www/fall2003/courses.html*

The relative-to-resource variety of URLs is especially useful when you are working with a collection of files that go together logically. If you use relative-to-resource URLs for your links, you can move those files as a unit to a different location, and the links between them will work without having to be updated.

On the other hand, relative-to-root URLs are useful when the resource to which the link is being made is **not** in the same directory as the file containing the link or one of that directory's subdirectories.

That said, when a browser encounters a relative-to-root URL (the type that *begins* with a slash), it constructs a complete URL using only the protocol identifier and host name from the URL of the file containing the link, appending those to the relative URL.

For example, if the page containing the link has this URL

http://*www.cs.cmu.edu/afs/cs/user/jar/www/index.html*

and the relative-to-root URL consists of

*/courses.html*

the browser would construct the following complete URL:

http://*www.cs.cmu.edu/courses.html*

Finally, one very important thing to say before we move on to our next topic: It is this last variety of relative URL—the relative-to-root variety—that you, as an SSD1 student should always use for your work in this course. Incidentally, using relative-to-root URLs will be especially important later in the course when you begin working with servlets and with the iCarnegie Servlet Workbench.

**Relative-To-Root vs. Relative-To-Resource Demo**

Before we move on to our next topic, let's try a demonstration that will illustrate vividly the difference be a relative-to-resource URL and a relative-to-root URL, a difference that can amount to the presence or absence of an initial slash (/).

Listings 1 and 2 below show the code of two HTML pages. Each of these pages contains an **IMG** element, and each of these elements refers to identically named images. The only difference between the two **IMG** elements is this: the first contains a relative-to-root URL, and the second, a relative-to-resource URL. Applying what we've just learned, the **src** attribute of the DemoRelToRootURL.html's **IMG** element specifies a relative-to-root URL because the URL begins with a slash (/), not with a server name. Also, since there are no other directories specified before the file name, the file *test.jpg* is said to be in the server's root directory. In contrast, the **src** attribute of the *DemoRelToResURL.html*'s **IMG** element specifies a relative-to-resource URL. The reference both lacks a server name and does not begin with a slash (/). Also, since there are no other directories specified before the file name, the *test.jpg* is said to be in the same directory as the page containing the link.

The next step will demonstrate just where the system expects to find the image files. Copy and paste sample HTML pages shown in Listings 1 and 2 into a plain text editor, such as Notepad, and save them each to your Desktop as files with .html extensions. Now, right-click the following link and save a copy of the test.jpg to your Desktop and a copy to your computer's root directory (which is probably C:\): [test.jpg](http://www.icarnegie.com/content/SSD/SSD1/2.1/normal/pg-www/pg-whats-in-the-web/pg-links-to-othr-docs/test.jpg" \t "externalWindow). (Note: for the purposes of this demonstration, test.jpg need not actually exist on your computer; the demonstration will work without it, so you can skip the last step if you want.)

|  |
| --- |
| <!DOCTYPE HTML PUBLIC '-//W3C//DTD HTML 4.0 Transitional//EN'> <HTML> <HEAD> <TITLE>Demonstrate Relative-to-Root URL</TITLE> </HEAD> <BODY> <IMG src='/test.jpg'> </BODY> </HTML> |
| **Listing 1** *DemoRelToRootURL.html* |

|  |
| --- |
| <!DOCTYPE HTML PUBLIC '-//W3C//DTD HTML 4.0 Transitional//EN'> <HTML> <HEAD> <TITLE>Demonstrate Relative-to-Resource URL</TITLE> </HEAD> <BODY> <IMG src='test.jpg'> </BODY> </HTML> |
| **Listing 2** *DemoRelToResURL.html* |

Once you've created these two HTML files on your Desktop, open *DemoRelToRootURL.html* in Internet Explorer, and what you'll see will be something like what is shown in Figure 1. Now, right-click on the image in the page, and then select the **Properties** command.

When the **Properties** dialog box appears (Figure 3), the image's expected location in terms of a path and file name is displayed after the label **Address**. Of course, the path and file name "C:/test.jpg" is equivalent to the Windows path and file name specification "C:\test.jpg"; and, as Figure 3 shows, the browser expects to find test.jpg in the computer's root directory.

If you are working with Netscape, you can accomplish this demonstration through slightly different steps. Annoyingly, those steps may vary, depending on which version of Netscape you are working with. So, by way of illustration, I walk through a Netscape demonstration with the version I am using: Netscape 7.02.

First, I open the page with Netscape, and, for some reason, Netscape 7.02 displays a blank page, not the graphic. However, I right-click the blank page, and then select the **View Page Info** command on the menu that appears, as shown in Figure 4.

When the **Page Info** dialog box appears, I click the **Media** tab, and the image file's location ("file:///test.jpg") is displayed both in the **Address** column and after the label "URL:" (Figure 5). Again, the URL "file:///test.jpg" is the Netscape equivalent of saying that it expects the file to be in the computer's root directory. Since Netscape 7.02 doesn't display the image, I must confirm that the URL constructed by Netscape from the relative URL in the link, is in fact pointing to the file in the machine's root directory. So, I select and copy the URL reported by Netscape (Figure 6) and then paste into the **Open** box of the Window's **Run** dialog (Figures 7 and 8).

When I click **OK**, test.jpg is opened in the default JPEG application, which on my machine is Internet Explorer (Figure 9). Notice the value displayed in the **Address** box.

Now, go through the process outlined above with *DemoRelToResURL.html*: open it in a browser and view the image address information. Figure 10 shows that the relative-to-resource URL tells the browser to look for test.jpg in my Desktop directory, which is the same directory where *DemoRelToResURL.html* is located.

To summarize this demonstration, the presence or absence of a slash (/) at the beginning of a path specification marks the difference between a relative-to-root URL and a relative-to-resource URL, a difference that cause the browser to look for the file in very different places.

**Using Complete and Relative URLs in a Web Page**

Below is a segment of code from the author’s home page (the file called *index.html*, which is in the *www* subdirectory). The segment illustrates how two *links* have been created. *Links*, by the way, are elements in a Web page that users can click to be "taken" to a different Web page. In Web jargon, they are said to be "following a link." The link shown below contains the relative URL to Jim’s freshman advisor page. The URL in the link below is a relative URL because it omits the protocol identifier and host name, and because it refers to a file that is also in Jim's *www* subdirectory, it also omits the path information, making it a relative-to-resource URL.

<A href='FA.html'> Freshman Advisor</A>

However, the URL in this next link refers to the "iCarnegie: In the News" page and is a complete URL:

<A href='http://www.icarnegie.com/about/news.html'> iCarnegie: In the News</A>

**Creating a Link to Other Files**

We got ahead of ourselves in the previous section, so some of this is going to be a little redundant. To create a link, you start with opening and closing anchor tags: <A> </A>.

Within the opening anchor tag, you add the HREF (**H**yperlink **Ref**erence) attribute like this: <A href> </A>. After the **href** attribute, you insert the URL within quotation marks. This finishes the link portion of the anchor.  After the opening anchor tag and before the </A>, you place the text you want to appear and represent the link on the Web page (that is, browsers will display the text, and readers of the page will be able to click it). Finally, you then add the closing anchor tag </A>.

Here again is the first link we encountered earlier in this page. Accompanying it here is an explanation of its various elements:

<A href='FA.html'>Freshman Advisor</A>

When a user clicks this link, the browser will fetch the file *FA.html* and display its contents. Since the path specified by the **href** attribute is a relative URL, the file identified by the path *must* be in the same directory as the page on which the link is located. The text *Freshman Advisor* will appear on the Web page, and mark the place users will click to access the file specified in the path. Most browsers will underline in blue text that is enclosed by opening and closing anchor tags (<A> and </A>)—in this case, *Freshman Advisor*. After a user clicks a link, a browser will change the text's color, usually to pink. Changing a link's color is meant to make it apparent to the user that he or she has visited the link.

Here again is the link to the "iCarnegie: In the News" page. Since this page file is not in Jim’s www subdirectory, a complete URL must be used.

<A href='http://www.icarnegie.com/about/news.html'>iCarnegie: In the News</A>

**Creating an Internal Link to a Place on the Same Web Page**

You can also create links to other places in the same page. To do this you need two tags. The first one is an anchor tag with the **name** attribute. This literally gives a name to a point in the document that becomes the *target* for the link. To create a target like this, you can write something like following:

<A name='office'><H2>Jim's Office</H2></A>  
 *Text about the office goes here*

Now that the target *office* has been created, you can create a link somewhere else in the document that will cause browsers to jump to the place named *office*. That link might look something like this:

One of our offices is <A href='Jim.html#office'>Jim’s Office</A>.  
Feel free to explore other offices as well.

Think of the hash ("*#*"**)** as a tab within the *Jim.html* file. The hash tells browsers to display the section of Jim.html where the *office* target is found.

**Embedding an Image into Your Web Page**

There are many issues involved with adding images and text, and aligning those in different ways. While much of this is beyond what we want to do here, simply embedding a picture in a page is fairly straightforward.  First, you'll need to learn about a new element, the **IMG** (Image) element.  It has a required attribute, **src**, that provides the URL of the image to be included. If, for example, you want to display a picture of downtown New Orleans on Mardi Gras night and you have a copy of the picture file in your www folder, you can use a relative URL as the URL to put the picture in your Web page. Therefore, since Jim’s picture file is in his www folder, he can use a relative URL for the picture file's URL. The following HTML statement puts his picture on the Web page:

<IMG src='jim.jpg' align='left'>

The align='left' attribute puts the picture on the left side of the page.

**A Little Extra to Make Data in Your Web Page Easier to Read — Tables**

There are two ways to display information in a tabular manner. One is to use the tags <PRE> and </PRE>. This is not always ideal since each browser may do the display a little differently. Another way to display tabular data is to use a table. Tables might seem confusing initially, but after you've studied the syntax and built a few, they shouldn't seem too hard. A drawback of tables is that there is a lot to their syntax, much more than for any other HTML tag we've studied so far.

In the following <TR> stands for table row, <TH> stands for table header (headers are usually on the first or top row), and **<**TD**>** stands for table data. Here is the basic table syntax:

<TABLE>  
<CAPTION> *text of caption goes here* </CAPTION>  
<TR><TH> column 1 name </TH><TH>column 2 name </TH>… </TR>  
<TR><TD> row1, col1 data </TD><TD>row1, col2 data </TD>… </TR>  
<TR><TD> row2, col1 data </TD><TD>row2, col2 data </TD>… </TR>  
<TR><TD> row3, col1 data </TD><TD>row3, col2 data </TD>… </TR>  
…  
</TABLE>

To create consistency, each row must have the same number of columns.

You can place the caption in different places with the **align** attribute. It defaults to the horizontal center at the top of the table. You can align it at the bottom of the table like this:

<CAPTION align='bottom'> *text of caption goes here* </CAPTION>

You may also wish to use the **align** attribute with the table row tag <TR>. The **align** attribute can accept the values **left**, **right**, or **center**. Table rows typically default to **left** alignment. You can force **center** (or **right** with the appropriate change) like this:

<TR align='center'>

There are some other table attributes such as padding and borders that you may also want to explore.

**Finally**

This is the end of our look at HTML. We encourage you to continue your exploration of this topic and to continue to tinker on some practice Web pages. Once you've mastered the use of individual tags, try combining them—placing an image inside a table cell, for example, or creating an embedded image that also serves as a hyperlink. Enjoy!

## 1.2.5 Images and Multimedia

Next, we'll discuss the .jpg extension seen in the filename in the last section (jim.jpg). You will commonly find two extensions on image files: **.jpg** and **.gif**. The **.jpg** extension is used for **JPEG** (**Joint Photographic Experts Group**) files. JPEG files are compressed—which means that the data they contain is stored in a manner that reduces the amount of information needed to reproduce a picture. If the data was not compressed, JPEG files would be much larger and would take much longer to transfer over the Web—which is what must happen for the images to be displayed by a browser. Also, most browsers store image files on the user's hard drive, so that if the user wants to view an image again, the image can be displayed more quickly by being read from the drive rather than transferred again over the Web. This is an example of a process known as *caching*, and caching provides yet another reason for compressing image files—the larger an image file is, the more room it will take up on a hard drive.

The **.gif** extension is used for **GIF** (**Graphics Interchange Format**) files.  Like JPEG files, the data in GIF files is compressed.  The major difference between the two formats lies in the way compression is achieved.  The data of GIF files is compressed in such a way that none of the original image information is lost. However, GIF files are limited in the range of colors they support. JPEG files, on the other hand, support more colors, but some of the original image information is lost during compression—a fact that can have an impact on image quality. Therefore, most Web masters recommend using JPEG compression for photographs and GIF compression for other kinds of graphics, such as icons and drawings. It's also important to note here that you do not need to know a lot about compressing or decompressing to use of either of these formats: most graphics packages and browsers handle these processes automatically. For example, if you are working on a photo in PhotoShop, all you have to do is save the photo as JPEG file (an option available through the **Save As** dialogue), and PhotoShop compresses it in JPEG format automatically. Browsers will then automatically employ the correct decompression technique to display it. The same is true for GIF files.

If you read articles or books on building Web pages, you will find disagreement about the use of JPEG and GIF files. Transferring these kinds of files over the Web does take time, and if your page has a many of them, users viewing your page will pay a price in terms of time. Therefore, use these types of files judiciously and only when pictures and graphics will enhance your Web page; do not use them just for the sake of using them. The same advice holds for audio and video files (both of which are beyond the scope of this class).  If you use these types of components wisely, you will create superior Web pages.  However, if you use them arbitrarily and without much forethought, you can end up with pages that are flashy, annoying, and a hodgepodge of confusing and useless information.

**1.2.6 Organizing the Process, Validation, and DOCTYPE Elements**

**Steps of the Process**

There are several steps that you can follow to make this process very organized and rewarding, as well as to reduce the total overall number of errors and amount of frustration:

1. Outline your Web pages in a logical manner, based on subject matter—do not worry about appearance initially.
2. Using paper, lay out each page according to the outline developed in step 1. Do this with just text at first—do not worry about graphics.
3. Mock up the page with big print for headers and with lines for the border, to get an idea of the overall look.
4. Identify the links you will need on the page.
5. Find several friends or classmates and ask their opinion and suggestions for improvement.

Once this is done, go to your computer and follow the procedures your teacher outlined for creating Web pages. When you're ready to type the actual HTML, follow these steps to build your Web page:

1. Work from top to bottom, adding one element at a time. After you add the first element, save the document and examine the page in a browser. If it doesn't appear correct, you may have a problem with one of your tags—find it and fix it. If you do not like the style that you chose, adjust it.
2. Add the second element and debug it in a similar manner.
3. Continue until you have the formatting and text in place. DO NOT PUT IN GRAPHICS OR LINKS YET.
4. Find those friends or classmates again and get their advice and suggestions.
5. Make the needed changes and test it in a manner similar to what you did before.

For the present, leave out the links and repeat this process with your other Web pages. Once you have your Web pages finished in terms of the text and formatting, you can add the links. Add the links one at a time and test each link individually. When you have all the links in, find those friends or classmates yet again and get their final opinion.

In summary, the process of Web page creation involves four steps:

1. Define the Web page content
2. Plan the look of the page and the links needed
3. Implement the Web page by writing it incrementally in small steps
4. Evaluate the Web page in two ways: a) test to see that it's HTML is correct and b) check to see that its appearance satisfies you and, as you have opportunity, other people.

Now, we've actually compressed a number of important matters into this last step, and in a moment, we will explore two of these matters in detail, but before we do that, here's a comment about how to go about steps 3 and 4. It is best to work through these two steps incrementally and iteratively. That is, you should build, say, the page you are working on, completing small parts of it, one at a time. Then, once you've completed a part, go to step 4, and when you do, remember to solicit feedback from other people, as you have the chance. Then, go back to step 3 and complete another part. If you keep the parts you build in step 3 small and cycle through steps 3 and 4 often until you are finished, you will find that the amount of time you spend on the project will be less than the amount you would have spent if you had built the entire page before evaluating it. Also, as you change your page, be sure to use the "Reload" button (if you're using Netscape) or the "Refresh" button (if you're using Internet Explorer) each time you go to view the page: doing so will ensure that your browser displays the latest version. Browsers store copies of pages in their cache, and a reload/refresh command will tell the browser to retrieve a fresh copy of the file to display rather than using a copy it may have in its cache.

Once you are done, you can add your graphical elements. See this module's next page [1.2.7 Some Final Thoughts](javascript:ContentByName('pg-final-thoughts');) for a few comments about the use of graphics.

**Verification, Validation, and HTML Validators**

In many professional circles these days—from software development to engineering and design—you’ll hear the watchwords “verification and validation.”  And, while you don’t need to remember these terms, you do need to remember the questions that are behind them. The question behind the term *verification* is this: "Is the product that has been made what the specifications ask for?" For this course, that translates into, “Is the HTML page that has been created what the specifications (the exercise or exam instructions, for example) ask for?” Stated this way, it is easy to see that this type of verification will play an important role in success in this course.

Behind the term *validation* is a question might be stated this way, “Has the product been made in a way that conforms to real-world conventions and standards for that type of product?" Now, that may seem a little abstract, but in many areas of life (medicine, construction, engineering, and so on), these real-world conventions and standards are widely-recognized and published, and they have come into being for some very important reasons—safety, security, ethics, usability, interoperability, backward compatibility—to name just a few.  But, that is still somewhat abstract for our purposes. However, fortunately for us, in this course the question behind validation is very specific one, and one that is easy to understand. For us, that question becomes, "Does an HTML page conform to the syntax and grammar published in the appropriate HTML specifications?" Also, fortunately for us, there are tools that can help us answer this question quickly.

That is, part of what you want to accomplish in step 4—checking that your page's HTML source is correct—can be greatly aided by the use of a tool known as an *HTML validator*. A validator is a tool that checks source code for not only the gross errors that even the most forgiving browsers catch but also of violations of various published conventions. HTML has evolved over the years and some new elements and conventions have been introduced while others have become obsolete. HTML validators help us determine if HTML code conforms to a particular published specification: HTML 4.01, HTML 4.0, or HTML 3.2, for example. When you work with Java later in this course, you will come across a similar step, called *compilation*, in which, among other things, the syntax of a servlet's Java code is validated. HTML validation is an equivalent step for HTML.

Why use an HTML validator instead of a browser to check your page's source? Because most recent Web browsers can be quite "forgiving"—that is, they can "fix" a variety of HTML errors before they render a page, hiding the problem from the user. This wouldn't be so bad for page authors such as yourself—if you could assume that all people would read your page using the most current browser and that all of the most current browsers, regardless of their brand, were equally forgiving. But, these are assumptions you cannot make. The capacity of browsers to "forgive" HTML errors varies from brand to brand and from version to version of the same brand—later versions typically being more forgiving than earlier. But, the situation is even more complicated: there other applications besides browsers that retrieve and render HTML pages, such as plug-ins and media players. So, as an author of an HTML page, you need to take such situations into account. But, beyond these, there's another consideration. HTML pages that conform to a published specifications are not only more likely to work with a wide variety of software that retrieves and renders Web content, they are more likely to continue to work properly, even as HTML and Web technologies evolve.

**The WDG HTML Validator**

The WDG HTML Validator is one of the HTML validators you will encounter in this course. It is one published by an organization known as the Web Design Group (WDG) and is available at their Web site. You don't need to visit that site just now, but when the time come, click the following link and the Validator's page will open in a separate browser window: [WDG HTML Validator](http://htmlhelp.com/tools/validator/upload.html" \t "externalWindow).

Figure 1 shows the validator site, as you will see it when you visit it. The procedure for using the validator is straightforward. Click **Browse...** and browse to the file you want to validate.

When you locate the file, in the **Choose file** dialog box, select the file and click **Open**.

When you click **Open**, the validator loads the file, and the file's path should be visible in the validator's text box, as in Figure 3. Then, select the **Include warnings** and **Show input** check boxes if they aren't already selected, and click **Validate it!** Selecting the **Include warnings** and **Show input** options will cause the validator to display both errors and warnings and a listing of the file evaluated in its "HTML Validation Results" report.

<HTML>  
<BODY>

<STRONG><EM>A common problem: Incorrect nesting</STRONG></EM>

A misplaced element:  
<!DOCTYPE HTML PUBLIC '-//W3C//DTD HTML 4.0 Transitional//EN'>

A missing slash:

<FORM>Form element goes here.<FORM>

</BODY>  
</HTML>

**Listing 1** *HTMLerrors.html*

Listing 1 shows the source of the file used in the validator demonstration shown in the figures in this page, and as you can see, HTMLerrors.html's source is listed in a section of the "HTML Validation Results" report name "Input" (shown in Figure 4). (Incidentally, the "Input" section is at the end of the report, after the "Document Checked" and "Errors and Warnings" sections.)

|  |  |  |
| --- | --- | --- |
|  |  | |
| **Figure 4** *The "Input" section of the validator's "*  *HTML Validation Results" report* | **Figure 6** *A WDG HTML Validator help page* |

Figure 5 shows an example of the "Errors and Warnings" section. This is the result of selecting the **Include warnings** option.

In the validator report's "Errors and Warnings" section, all HTML elements are hyperlinked to the validator's help page for the hyperlinked element, as shown in Figure 6. These help pages contain relevant information about the particular elements, such as their syntax, attributes, and status within the HTML specification against which the page is being evaluated.

Also hyperlinked are the line numbers for the lines of source in which the validator identifies issues. These hyperlinks are part of the entries in the report's "Errors and Warnings" section. These entries are bulleted, and each entry consists of the hyperlinked line number of the line of source where the problem is—and the number of the character in the line where the problem is located. Below that is a fragment of the line of source, with a caret (^) underneath the character that is identified by the character number. Below these may appear either a description of the problem found (what starts with "Error:"). Sometimes a problem causes the validator to generate more than one entry. In these cases, the validator provides additional information in the second entry that supplements what is in the first entry. For an example of this, see Figure 7, the entries labeled "Line 4, character 56" and "Line 4, character 9." What is important here is that, the validator tries to diagnose problems, and, although its diagnoses aren't always correct, it usually provides enough information for you to make an accurate diagnosis.

|  |  |
| --- | --- |
|  |  |
| **Figure 7** *The "Errors and Warnings" section, lines 4 to 7* | **Figure 8** *The "Errors and Warnings" section, lines 7 through 13* |

Now, take a moment to look back over Listing 1 and Figures 5, 7, and 8. The file used in this demonstration contains some common HTML problems: improperly nested tags (Figure 7, line 4), a misplaced element (Figure 5, line 1 and Figure 8, line 7), and an end tag that is missing a slash (/) (Figure 8, lines 11 and 13). Take a close look at the lines of source and at how the validator reports the errors they contain. Doing this will help you become familiar with these common errors—and how the validator handles them and others like them.

This brings us to the end of our introduction to the WDG HTML Validator. Take a few moments now to visit the site: [WDG HTML Validator](http://htmlhelp.com/tools/validator/upload.html" \t "externalWindow). While you are there, be sure to try the Validator out a few times. You also might find it useful to read some of the supporting pages linked to the validator's front page such as "Common validation problems," "Tips on using the validator," and "4 reasons to validate your HTML."

When you return, we'll move on to our next topic, and as we do, take another look at the "misplace element" problem shown above (see Figure 5, line 1 and Figure 8, line 7). That problem involves the **DOCTYPE** element, which is what we'll learn about next.

**The DOCTYPE Element**

The validator flagged the **DOCTYPE** element in HTMLerrors.html's line 7 because **DOCTYPE** elements are supposed to be on the very first line of a file, right before the beginning <HTML> tag.

The **DOCTYPE** element's purpose is to tell agents that process Web files—such as validators, Web browsers, and the like—what HTML specification is followed in the file's source. The **DOCTYPE** element we encountered earlier in HTMLerrors.html is:

<!DOCTYPE HTML PUBLIC '-//W3C//DTD HTML 4.0 Transitional//EN'>

Breaking this element down into it parts, this particular element declares the file to be an HTML document that follows the HTML 4.0 Transitional specification. This means that source in the file largely conforms to the HTML 4.0 specification but may include certain elements that are technically obsolete in the specification but are permitted in a transitional edition of the specification.

We can't go into more details here. What is important to say here for the purposes of this course, is that you must use this exact **DOCTYPE** element in all of your HTML files for this course, here on out. Now that we've introduced the validator, you will be required to use it to validate all of your HTML source, and your HTML files will not pass the validator without an appropriately placed **DOCTYPE** element.

## 1.2.7 Some Final Thoughts

You may have figured out that the author is not a big fan of cluttering up a Web page with lots of graphics in the way of JPEG and GIF files. There's a reason for this: graphical elements should enhance a Web page and not detract from the information presented. Also, remember that JPEG and GIF files take time to download, and if people are trying to access your Web page at a very busy time, the more images your page has, the longer it will take people to download it. Therefore, since some very simple HTML can produce some very nice formatting, be sure to make the most of HTML's capabilities and limit your use of graphics to something that is reasonable.

Also, make sure that you are not using the copyright or intellectual property of someone else without permission. This is very important.

Finally, if you are going to reduce the size of a picture, do so with software that allows you to reduce the size of the picture as well as the size of the file. JPEG files are compressed, but they can still take up a lot of space. If you just display a 4x5 picture as a small thumbnail picture, you still transfer the full data file—no time is saved by just displaying the file in a smaller format. But, if you edit the file with software, you can reduce the actual size of the file as well as the picture.

This has been a brief overview of HTML and the Web. It is supposed to get you started, so it is just a beginning. You can write very useful and pleasing Web pages without going beyond the material presented in this unit. If you do that, we will be very satisfied. However, we encourage you to continue to explore HTML and its capabilities throughout the term and we hope that the suite of Web pages that you will prepare for this module will continue to grow and evolve as you acquire more skill and knowledge.

**1.3 Introduction to HTML Forms and Servlets**

We begin this module by introducing forms and servlets in the context of a conceptual problem—the need for dynamic content and information submission and processing—teaching forms and servlets as two language-specific pieces to a solution to this problem. We then move on to take a detailed look at the **FORM** element and a basic set of form controls. Next, we take a high-level look at Java servlets and the role they play in extending the basic client-server model. With this background, we then turn to the practical, looking to gain experience in creating and submitting forms to a supplied servlet. To this end, this section will introduce the iCarnegie Servlet Workbench tool and provide some experience using it.

**1.3.1 Working with HTML Forms**

**The Need for Dynamic Content**

So far we have seen HTML primarily only in its role as a markup language—a tool that allows us to format and present content to users who view it with Web browsers. That’s a very good start, and we should be pleased with the ability we've acquired so far. Now, so long as our Web pages meet our users' every need and are linked in such a way that users find what they want easily, we are in good shape.

However, a bit of reflection will quickly show that our position is still pretty limited. Several factors might complicate things and make meeting user needs with ready-made, static pages intractable. Sometimes, the needs of our users may outstrip our ability to meet them. For example, if our users need different pieces of our content to be combined in different ways, or if the content on our pages changes frequently, or if our users need the content we supply to be combined with information they supply. We can't create new pages or revise existing pages fast enough to keep up with demand. Fortunately, this problem and its solution aren't new to data processing. HTML has a provision for one part of the solution: the **FORM** element. The other part of the solution, the part on the server side, involves Java servlets. Servlets will be discussed in more detail in the following page.

In this section, we will cover the basics of the **FORM** element and learn how to submit and process forms. Beyond this are a number of topics that are outside the scope of this course, but before concluding this section, we will take a moment to identify these topics for further study and point you to sources where you can read more about them.

**The FORM Element**

The FORM element is a container element that creates a special area on an HTML page known as a *form.* A **FORM** element can contain, in addition to the usual page content, markup tags, and special elements known as *controls* or *widgets*. It is these special control elements that allow users to interact with the form. Users can modify control values by entering text, choosing items from a menu of items, and so forth. When the user is finished, one of these controls allows them to submit the form to an agent, such as a Web server, that will then process the form. After we cover the rudiments of the **FORM** element, we will cover a set of the most important control elements: the **INPUT** and **LABEL** elements, the **BUTTON** element, and the **TEXTAREA** element.

Like many HTML elements, the **FORM** element consists of a pair of beginning and ending tags: <FORM> and </FORM>. Between these two tags is specified the content of the form, which can be typical page content, other HTML markup tags, and form controls:

<FORM *element attributes*>

form content ...

</FORM>

In addition to specifying the form's content, the **FORM** element specifies the program on the server that will process the form when it is submitted and the method by which the form data will be sent to the server. This information is specified as attributes that are entered in the beginning <FORM> tag. The **action** attribute designates the program that will process the form data; the method attribute specifies the HTTP method by which the data will be sent to the server:

<FORM action='http://example.com/servlts/myprog' method='post'>

form content ...

</FORM>

As shown in the above, the **action** attribute provides the name of the program that will process the form data. The program name is specified within single quotation marks (') and can include a complete path name, including a site URL. (**Note**: Using double quotation marks (") are also acceptable and actually more common. However, for reasons we can't get into just now, using single quotation marks will help you keep matters less complicated than they might otherwise be when you get to Unit 2.) The **method** attribute has only two valid options—**post** and **get**. We won't get into how the two differ at the moment. We will only be working with the post method for the moment.

The **action** and **method** are the only **FORM** attributes we will work with in this course. But, be aware that **FORM** element takes a number of other attributes—the **enctype**, **accept**, **name**, **title** attributes and others. In addition, it also features several "event" attributes— for example, some of the more commonly used are the **onsubmit**, **onreset**, **onclick**, **ondblclick**, **onmouseover**, and **onkeypress** event attributes. These attributes allow one to assign scripts to be executed in response to a specified "event," such as a key press or mouse click.

**Form Controls and Control Types**

There are a few basic types of form controls that we will introduce immediately and cover in some detail, such as *button*, *checkbox*, *radio button*, *label*, *menu*, *file select, and text input controls* (including *password*). Beyond these are a few other special-purpose controls—*hidden, image*, and *object controls*—which we will be satisfied with merely mentioning: they are beyond the scope of this course. The number of form elements used to create these controls is somewhat smaller. Some elements can create more than one type of control.  In fact, the all-purpose **INPUT** element can be used to create all but the more special-purpose controls—such as menu controls.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **buttons** | | **checkboxes** | **radio buttons** | **label** |
|  | |  |  | My label text |
|  | |  |  |  |
| **menus** | **file select** | **text input** | |
| single option  scrollable options grouped options |  | single-line input  multiple-line input (text area) password | |

**Quick facts about controls.** Let us now consider a few basic facts about controls in general that makes understanding the nature of each type of control a little easier.  First, each control has a *control name*, which is designated by its **name** attribute.  This control name allows the control to be referred to by other client-side programming entities, such as scripts, and the scope of a control name is the **FORM** element in which the control resides. But, most important for our current discussion is that the control name plays an important role in the form submission process, which we will say more about shortly.

Second, all controls can have both *an initial value* and a *current value*. The initial value (also known as the *default value*), if one is specified, is designated by the control’s **value** attribute, though there are a couple of exceptions to this (more about what these exceptions are when we get to discuss the syntax of specific control elements).  A control’s initial value never changes. In contrast, the defining characteristic of the current value is that it **can** be modified.  The current value is set to the initial value when the page is first loaded.  After that, current values can be modified, by user actions.  Of course, when a form is *reset*, current control values are once again set back to their initial values, where initial values are specified.  By the way, controls that have no initial values specified are considered *undefined* when their current values have not been modified—either after the form is first loaded or after it is reset. Controls that remain undefined cannot be sent with a form to a server. But, the term *undefined* may carry connotations that don't apply here, so let's probe into what it means in this context, and as we do, be aware that the phrase "remain undefined" in the preceeding sentence is an important one and was used deliberately. *Undefined* is used to describe situations for which no standard has been defined about how user agents (browsers, for example) are to behave in those situations; therefore, how browsers handle (or don't handle) those types of situations may vary from browser to browser. For example, some browsers when they process forms and encounter certain types of controls that have no current or assigned initial (default) values might assign default values on their own (such as an empty string when a string data is called for) and then include controls and values in the data set to be sent to the server. Other browsers might ignore the controls and exclude them from form data sets. The important point to remember here is that *undefined* means you can't predict browser behavior, and, as a developer of Web page, that is something you want to avoid.

Finally, when a form is submitted for processing, the controls that have values (including those with browser-assigned default values) have their names and values combined into *name-value pairs*, which are then submitted to a server with the form.  All controls submitted with a form, are known as *successful controls*; as mentioned before, controls that remain undefined are not considered successful controls, and are not sent to a server. But, like the word *undefined*, the words *successful* and *unsuccessful* carry certain connotations that do not necessarily apply here. Unsuccessful controls are merely controls that, for whatever reason, do not qualify to be included in a data set sent to a server, and that isn't necessarily a bad thing. Why should controls be sent to the server if there are no values to submit with them?

* *Buttons*. There are three types of buttons: *push button*, *reset buttons*, and *submit buttons*. Push buttons are beyond the scope of this course, so we won't say anything more about them here. That said, reset and submit buttons have dedicated functions. Reset buttons, as their name suggests, cause the controls of their host form to be reset to their initial values. Submit buttons cause the form to be submitted to the program identified in the form’s action attribute. By the way, a form may have more than one submit button.
* *Checkboxes*. Checkbox controls allow users to toggle them “on” and “off” (with a click of the mouse or a press of the SPACEBAR).  When a checkbox is “on”—that is, when it is “checked”—the control is considered "successful" and its name-value pair is submitted with the form; otherwise, it is **not** considered a successful control, and its name-value pair is **not** submitted with the form.
* *Radio buttons*. As do checkboxes, radio buttons allow users to toggle them “on” and “off” (again, with a click of the mouse or a press of the SPACEBAR). However, in contrast to checkboxes, radio buttons are usually used in groups of more than one (though this is a matter of practice, not of syntax requirement).  A radio button group is created by assigning all of the radio buttons that are to be in the group the same control name. The effect of creating a radio button group is to make the buttons in that group mutually exclusive: only one button can be in the “on” state at a time.  When one button is clicked—that is, turned “on”—the button that was “on” prior to that is set to “off.” As with checkbox control, when a radio button is “on,” it is considered "successful" and its name-value pair is submitted with the form; otherwise, it is **not** considered a successful control, and its name-value pair is **not** submitted with the form. The implication this has for radio button groups is that only one of its button name-value pairs is submitted with its form.
* *Labels*. Label controls are used to specify label text for form controls that do not have provide for labels in their own syntax, say, by way of an attribute. Examples of this type of controls are checkboxes and radio buttons, and we'll say more about them and label controls later in this page.
* *Menus*. So-called menu controls allow users to choose from set of predetermined options. However, form menu controls are unlike the controls that are referred to as "menus" in the Windows and the MAC environments—such as the bar across the top of a window displaying options like "File," "Edit," "View," "Insert," and so on or the pop-menu that appears after a right click. Rather, form menu controls are basically what are also known as "combo boxes" and "drop-down lists," and form menu controls come in three basic formats: single-option drop-down lists, scrollable-option list (which features the ability to allow for the selection of single or multiple options), and grouped-option drop-down list.
* *File select controls*. File select controls allow users to browse to files, select files, and submit the contents of a file or files with a form.
* *Text input controls*. There are three types of controls that allow users to input text: one for single-line input, one for multiple-line input, and one for text that should not be displayed as it is being typed—passwords, for example.
* *Image controls*. Image controls are submit buttons; however, instead displaying a textual button label on their face, image buttons display a graphic image.

**The INPUT Element**

The **INPUT** element is a general-purpose element that can be used to create most of the controls introduced earlier in this page*.* The controls that the **INPUT** element can create are password and single-line text input controls; checkbox and radio button controls; submit, reset, and push button controls; and file select, image, and hidden controls. What are left out of this list are object, menu, and multiple-line text input controls. Along this line, it’s also worth mentioning here that, although the **INPUT** element can create all three types of buttons, its facility for doing so is limited, and for situations that require a richer set of options, the **BUTTON** element is a better choice, which we'll cover in more detail later in this page.

The **INPUT** element consists of only one tag: <INPUT *element attributes*>.  Its *element attributes* include the following:

* Attributes mentioned earlier in the quick facts about all controls : the **name** and **value** attributes and a set of “event” attribute
* Attributes dedicated to its function as an element— principally the **type** attribute
* Attributes that limit a control's input (the **checked**, **disabled**,  **readonly**, **size**, and **maxlength** attributes)
* Attributes that we'll say a little more about later in this page: the **tabindex** and **accesskey** attributes

The **INPUT** element’s **name** and **value** attributes function basically as described earlier. The **name** attribute specifies the control name. Using the control name, the form submission process allows other client-side programming entities, such as scripts, to access the control. The **value** attribute allows page authors to provide a control with an initial or default value, though an initial value is not required for most control types. The only exceptions to this are radio button or checkbox controls. With the radio button or checkbox controls, the **value** attribute designates the value that is to be submitted if the control is "on"—that is, if it is checked.

The **type** attribute, on the other hand, is the **INPUT** element’s principle attribute, controlling what type of control is created. Options for the **type** attribute include **text**, **password**, **checkbox**, **radio**, **file**, **hidden**, **submit**, **image**, **reset**, and **button.** These options are described in more detail below (except for the **hidden** option, which is beyond the scope of this course):

* ***text***. This **type** attribute causes an **INPUT** element to create a single-line text input control. This is the default control type if no **type** attribute is specified. The **value** attribute, if present, specifies the initial value for the text input control, which will appear in the control when the form loads. The **size** attribute specifies the visible width of the control on the form. For text input controls, this width is specified in terms of number of characters (for other controls, it is specified in terms of pixels). The **maxlength** attribute sets a limit on the number of characters that can be entered into the control. When this limit is reached, the control generates an audible beep.

|  |  |
| --- | --- |
| **rendered control** | **source** |
|  | <input type='text' name='txtline'> |
|  | <input name='txtline2'> |
|  | <input name='txtline3' type='text' value='Default Value'> |
|  | <input type='text' name='txtline4' size='12' maxlength='200'> |

* ***password***. This **type** option creates a single-line text input control that obscures from view with a special character, such as an asterisk (\*), the characters that are entered into it. This characteristic, of course, makes the control a suitable choice when security is a concern, such as with password entry. In those situations, it is common to use it in conjunction with the **size** and **maxlength** attributes because passwords are typically limited to a small number of characters.

|  |  |
| --- | --- |
| **rendered control** | **source** |
|  | <input name='withoutdefault' type='password' size='8' maxlength='8'> |

* ***checkbox***. This **type** option creates a checkbox control. Unlike all other **INPUT** control types (except the **radio** type), **checkbox** types *must* include a **value** attribute, which specifies the value that is to be submitted with the form when the checkbox control is checked at the time the form is submitted. The control may be set initially to a checked state with the **checked** attribute. Doing so, specifies that, by default, the value of the **value** attribute will be submitted with the form—unless the control has been turned "off" in the meantime. The **checked** attribute is valid only for **checkbox** and **radio** **INPUT** types, and will be ignored if it is specified with any other control type.

|  |  |
| --- | --- |
| **rendered control** | **source** |
|  | **<name='chkbx1' type='checkbox' value='value-to-be-sent'>** |
|  | **<input name='chkbx2' type='checkbox' value='value-to-be-sent2' checked='checked'>** |

* ***radio***. This **type** option creates a radio button control. Unlike all other **INPUT** control types (except the **checkbox** type), **radio** types *must* include a **value** attribute, which specifies the value that is to be submitted with the form when the radio button control is checked at the time the form is submitted. As with **checkbox** control types, a radio button may be set initially to an "on" state with the **checked** attribute. Again, doing so specifies that, by default, the value of the **value** attribute will be submitted with the form—unless the control has been turned "off" in the meantime. The **checked** attribute is valid only for **checkbox** and **radio** **INPUT** types, and will be ignored if it is specified with any other control type.   
    
  Radio buttons differ from checkboxes in that they are commonly used in groups in which only one button can be selected at a time. **INPUT** controls of type **radio** are made into a group by assigning each control that is to be in the group the same value for its **name** attribute.

|  |  |
| --- | --- |
| **rendered control** | **source** |
|  | <input name='rdbtn1' type='radio' value='value-to-be-sent'> |
|  | <input name='rdbtn2' type='radio' value='value-to-be-sent' checked="checked"> |
|  | <input name='btn-grp' type='radio' value='default-value-to-be-sent' checked='checked'>  <input name='btn-grp' type='radio' value='alt-value'> |

* ***file***. This option creates a file select control, which enables users to select a file or files to have their contents submitted with a form.

|  |  |
| --- | --- |
| **rendered control** | **source** |
|  | <input name='selct-a-file' type='file'> |

* ***submit*** *and* ***image***. These options both create submit buttons. Recall that a submit button causes a form to be submitted for processing to the server agent specified in the **FORM** element's **action** attribute.   
    
   The **submit** option results in a submit button with a text label. If no **value** attribute is present, button's default label is "Submit Query." A value specified with the **value** attribute will override this default.  
  The **image** option creates a submit button that displays an image. The **src** attribute specifies the path and file name of the image to be displayed. When using the **image** option, it is good practice to specify alternate text with the **alt** attribute. Doing so will make the control accessible to users who rely on text recognition software to read interfaces to them. For this type of control, the software will read the text specified by the **alt** attribute. But, text specified with the **alt** attribute can benefit other types of users as well: the alternate text appears as a context tip when the mouse pointer is held over the control. **Note:** given the important roles alternate text can play, care should be taken to make the text as descriptive as possible, perhaps identifying the button's function as a submit button or otherwise, indicating its purpose. (By the way, specifying an **alt** attribute with an **image** **INPUT** control is not yet required for HTML 4.01 validation, as it is for the **IMAGE** element, but the principle is a still good one to follow even though it's not required yet.)

|  |  |  |  |
| --- | --- | --- | --- |
| **rendered control** | **source** | | **source for the equivalent button using the BUTTON element** |
|  | <input type='submit' name='subbtn'> | | <BUTTON name='subbtn' value='submit' type='submit'>Submit Query</BUTTON> |
|  | <input name='subbtn2' type='submit' value='Submit Answer'> | | <BUTTON name='subbtn2' value='submit' type='submit'>Submit Answer</BUTTON> |
|  | <input name='subbtn3' type='submit' value='Log On'> | | <BUTTON name='subbtn3' value='LogOn' type='submit'>Log On</BUTTON> |
|  | <input name='graphicsubbtn' type='image' src='tools.gif' alt='Submit Tools Information Form'> | | <BUTTON name='graphicsubbtn' type='submit'><IMG src='tools.gif' alt='Submit Tools Information Form'></BUTTON> |
| ***reset***. This option creates a reset button, which will set the current values of all form controls back to what they were when the form first loaded. If no **value** attribute is present in the reset button control, the button's default label is "Reset." A value specified with the **value** attribute will override this default. **rendered control** | | **source** | |
|  | | <input name='resetbtn' type='reset'> | |
|  | | <input name='resetbtn2' type='reset' value='Reset Form'> | |

**The LABEL and TEXTAREA Elements**

In addition to the **INPUT** element, two other elements are useful to our purposes in this course: the **LABEL** element and **TEXTAREA** element.

The **LABEL** element can be used to assign text labels or other information to controls that have no provision such labels in their syntax—checkboxes and radio buttons, for example. **LABEL** elements require both beginning and ending tags: <LABEL></LABEL>. These tags enclose the text that is intended to be the label's text. These tags may also enclose one control. When they do, that control is said to be associated with the **LABEL** element "implicitly."

|  |  |
| --- | --- |
| **rendered control** | **source** |
| Diamonds | <LABEL>Diamonds <INPUT name='jeweltype' type='checkbox' value='diamonds'></LABEL> |

But, **LABEL**s can also be associated with controls "explicitly." This is accomplished with the **LABEL** element's **for** attribute, which is set equal to the control's **id** attribute.

|  |  |
| --- | --- |
| **rendered control** | **source** |
| Ruby | <LABEL for='rub'>Ruby </LABEL> <input name='jeweltype2' type='checkbox' value='rubies' id='rub'> |

**The TEXTAREA element.** As we learned earlier, an **INPUT** element can create a single-line text input control, but often, what is needed is a control that allows for multiple lines of input, which is beyond the **INPUT** element's capacity. For that purpose, one must use the **TEXTAREA** element. Like **LABEL** element, the **TEXTAREA** element requires a beginning and ending tag. Within the beginning tag, page authors can specify **name**, **rows**, and **cols** attributes. The **name** attribute, as we learned earlier, specifies the control name. The **rows** attribute specifies the number of visible lines of text the control will display. The **cols** attribute specifies the control's visible width, in terms of average number of characters.

|  |  |
| --- | --- |
| **rendered control** | **source** |
|  | <TEXTAREA name='textarea' cols='40' rows='5'></TEXTAREA> |

**Form Design: Taking Users into Consideration**

Now that we have covered a basic set of form controls, there is one more important matter to mention before we move on. That matter is form design. In general, controls shouldn't be placed randomly onto a form without giving thought to how users might experience the form. Will users find the form easy to understand and to work with? They will if its controls are grouped logically, labeled clearly, and placed on the form in an order that they find to easy to work with. However, interface design is a large subject, and there isn't enough space in this course to say much about it. It will suffice for our purposes here to point out a few things that are worth mentioning in passing.

First, there are form control attributes that are designed to address some of these issues. These attributes are the **readonly**, **tabindex**, **accesskey**, and **alt** attributes. The **readonly** attribute can be used to allow users to see the contents of a control but not be able to change the contents, and there are situations when that is a very good thing to be able to do, from the standpoint of usability. The **tabindex** attribute is used to specify the order in which users can move the cursor or focus from control to control on the form using the TAB key. The cursor or focus moves first to the control with the lowest **tabindex** attribute and proceeds from there in order of ascending **tabindex** value (where "0" is the equivalent of "10"). The **accesskey** attribute allows page authors to assign "access keys" to individual controls. Access keys allow users to activate controls with the press of a key—as opposed to clicking them with a mouse. Finally, as we mentioned earlier in this page, the **alt** attribute can be used to controls accessible to, say, users who rely on text recognition software to read computer interfaces.

In addition to these, we learned earlier that forms can include other HTML markup tags, and one useful way to exploit this capability to enhance the appearance and design of your forms is to make use of the **TABLE** element. In this context, the **TABLE** element can be used to group controls on a form in a way that is logical or aesthetic. The following site has an example that uses a **TABLE** element to do just that. Visit [http://wdvl.com/Authoring/HTML/4/Tags/forms.html](http://wdvl.com/Authoring/HTML/4/Tags/forms.html" \t "externalWindow) and you will see both, an example of the form as it is rendered as well as the code that creates it.

**Topics and Resources for Further Study**

Our goal in this page has been to introduce you to the basic **FORM** elements that you will need in this course. However, the time will certainly come when you want to learn more, and our purpose in this section is to point you in the right direction—providing you with a short list of suggested topics and resources with which you might explore them, when that time comes.

Our suggested list of topics consists of the following: the **BUTTON** element; the **SELECT** element (for creating menu controls); the **LEGEND** and **FIELDSET** elements (which provide structure to forms); access keys, TAB navigation, and giving controls the focus; disabling and making controls read-only; and, finally, form content types. You can find out about all of these and more at the World Wide Web Consortium (W3C)'s very useful site. As an organization, the W3C is dedicated to promoting technologies that will help the Web achieve its potential, and the W3C site features a very useful write-up on the **[FORM](http://www.w3.org/TR/html401/interact/forms.html" \t "externalwindow)** [element](http://www.w3.org/TR/html401/interact/forms.html" \t "externalwindow). Be warned, however, W3C pages are written for a number of very technical audiences, and therefore, their specifications can be challenging to read in places. However, the rewards are usually worth the effort and patience expended.

As you already may know, in addition to the WC3 site, there a host of other HTML resource sites on the Web, and many of these include use write-ups on forms. Among these are those available at the Web Developers Virtual Library (WDVL): [http://wdvl.internet.com/Quadzilla/Tag\_Ref/form.html](http://wdvl.internet.com/Quadzilla/Tag_Ref/form.html" \t "externalWindow) and [http://wdvl.internet.com/Quadzilla/Tag\_Ref/](http://wdvl.internet.com/Quadzilla/Tag_Ref/" \t "externalWindow).

**1.3.2 Servlets, HTTP Connections, Server Push-Client Pull, and Refresh**

**An Overview of Servlets**

In our last section, we learned about the need for dynamic content and HTML's role in the client-side of the solution in the provision of the **FORM** element. In this section, we turn to the server-side of the solution and take a high-level look at Java servlets. Servlets are typically called from HTML forms (remember the **action** attribute), and form controls are the agency through which input is created for servlets. Let's learn a few more details about what servlets are and how they fit into the client-server model. Servlets are programs that run inside of complex software programs known as Web servers (hereafter referred to as "servers"). It is the servers that run servlets, in order to construct dynamic responses to client requests. Therefore, it should be clear that, although servlets respond to requests from clients, servlets do so indirectly—at the request of a server, not the client itself.

In this context, here is the basic sequence of events, in the order they occur. A user using a Web browser sends a *request* for information to a server. When the server receives the request, it determines if the request is for a static HTML page or for an application, such as a servlet. If the request is for a static HTML page, the server responds by serving the appropriate page to the browser. This typically is a file stored on the server's machine. However, if the request is directed to a servlet instead (incidentally, requests to servlets typically have URLs that contain the node “/servlet/”), the server's next action is to load and initialize the servlet—if the servlet isn't already running. Once the servlet is initialized and running, the server passes the browser request on to the servlet, which then performs some kind of process. When that process is finished, the servlet then sends its *response* to the server, which in turn forwards the response on to the client browser. This response is usually a fully formed HTML page that has been constructed by the servlet "on the fly." At this point, the servlet will continue to run, awaiting a subsequent request—after which it will continue to run. In fact, a servlet will continue to run indefinitely, though it can be terminated as a part of system housekeeping.

**HTTP Connections and Connection Life Cycle**

With HTTP client-server transactions as we have presented them thus far, the HTTP connection life cycle typically follows this pattern: an HTTP connection is opened, a client sends a request to a server, the server sends a response to the client, and the connection is closed. If the client then sends subsequent requests to the server, the same pattern holds—for each request, a connection is opened, the request is sent, a response is returned, and the connection is closed.

One additional concept should be introduced here with HTTP connections: user sessions. In the context of this course, a user session might be thought of as the series of interactions a user has with a server to carry out a user task, such as purchasing a book. The user session begins when the user first accesses the server through an initial HTTP connection, and the session ends when the user either completes or cancels the task (say, by clicking the Confirm Purchase or the Cancel Purchase button). Either of these events put an end to the session and to its series of HTTP connections.

Now back to typical HTTP connection life cycles, one thing to notice is that, because connections are both opened and closed for each request-response exchange, nothing we've mentioned so far keeps a server from seeing each request-response transaction as a discreet entity with no relation to any other transaction. Nevertheless, there are many situations in which a server *must* not only keep track of information from one transaction to the next, but also keep track of which client the information came from. To keep track of these details, servers drop cookies into the client browsers. A cookie is a piece of data generated by a web server and stored on the browser's computer for future reference. Cookies allow servers to "remember" information from one transaction to the next and to distinguish the requests that come from one client from the requests that come from other clients.

For example, consider an application that allows users to make flight reservations over the Web. Typically, such an application would lead users through a series of Web pages, each page being both the result of and the initiating point for a client-server transaction. At the beginning of this process, the application via a server might provide users with a page that allows them to specify information, such as desired destination, place of departure, and so on, as shown in Figure 1.

|  |  |
| --- | --- |
|  |  |
| **Figure 1** *Flights Search interface* | **Figure 2** *List of available flights* |

Once users submit the page (which is an HTML form) with the requested information, the server may in turn respond with a page listing all of the available flights that meet the specified criteria. This page would typically also include relevant information such as flight numbers, ticket prices, and so on (as shown in Figure 2), and the page would typically include controls that allow users to choose from the flights on the list.

Once users choose departing and returning flights, the server might provide them with a page that summarizes the details of their selected travel package, with the controls necessary to allow users to confirm and book the reservations, as shown in Figure 3.

|  |  |
| --- | --- |
|  |  |
| **Figure 3** *Book the roundtrip travel package* | **Figure 4** *Billing information* |

Once users choose to book the package, the server might respond with a page designed to collect the billing information necessary to make the actual reservations. For example, the page might request the customer's name, billing address, credit card number, and so on, as shown in Figure 4.

What is important here for our discussion of HTTP connection life cycle is this issue. When John Martindale—or any other user using the Customer Billing Information interface (Figure 4)—submits billing information, the server must then be able to combine that information with the correct flight information about the flights selected (which was submitted a couple of forms earlier in a separate HTTP transaction, see Figure 2). It must combine the correct information in order to be able to make the correct entries in the flight reservation database. The server is able to combine the billing information correctly with the flights selected by the same client, though data items were collected in separate transactions, by placing a cookie in the client's browser. Once a cookie is placed in a browser, it serves to identify requests and data with a particular client.

Without a cookie, a server has no way of knowing that two or more requests are part of the same user session. Notice that the Customer Billing Information form shown in Figure 4 contains no form controls containing information about the flights that have been selected—hence, when the form with the billing information is submitted to the server, the information about the flights selected will not be submitted with it. Thus, if after John Martindale submits his choice of flights to the server in Figure 2, Mary Zhu comes along and submits her billing information, Mary might mistakenly book (and be billed for) John's flights. Consider this scenario exclusively from the server's perspective, and the problem will be readily apparent. From the server's perspective, the sequence of requests it receives is as follows: a request to process a choice of flights (John's) and then a request to process billing information (Mary's). Without cookies to indicate which request belongs to which session, the server cannot know that the request to process a choice of flights came from John's session and the request to process billing information came from Mary's. Moreover, there are only two users in our imaginary illustration, and in a real life, during a peak period, a server like the one in our illustration might receive simultaneously literally hundreds of requests of different types from hundreds of different users. Without cookies, our server can't even distinguish between John's requests and Mary's requests, much less requests from hundreds of users.

By way of illustration, you can view the cookies that have been placed in your own browser. To view cookies placed in Internet Explorer, open an Internet Explorer window, click **Tools** on the main menu bar, and then select **Internet Options...** on the list of options that appears. On the **General** tab of the **Internet Options** dialog box, click the **Settings**. When the **Settings** dialog box appears, click the **View Files...** button, and an Explorer window will open to the Temporary Internet Files folder. Click the **Internet Address** heading, and cookie files should sort to the top. The file name of cookie files begin with the prefix "Cookie:"—as shown in Figure 5.

|  |  |
| --- | --- |
|  |  |
| **Figure 5** *Internet Explorer cookies in an Explorer window* | **Figure 6** *Netscape Cookie Manager* |

Viewing cookies placed in Netscape is a little more straightforward, open a Netscape window and click **Tools** on the main menu bar, and then select **Cookie Manager** on the list that appears, and then click **Manage Stored Cookies** on the submenu that appears. The **Cookie Manager** dialog box will then, showing the Stored Cookies panel (Figure 6). If you've used Netscape to sign on to this course, a cookie from icarnegie.com will be visible in this panel.

**Server Push–Client Pull**

Now, let us revisit the topic of dynamic content. So far, in our discussion of dynamic content and of static content for that matter we have seen the client-server interaction from the perspective of one that is driven by user action. That is, fundamental to HTML features like forms and hyperlinks is the idea that users make requests and servers respond to those requests, one for one. While this user-action-driven, response-for-request view of things is accurate overall, there are situations in which an alternative to this user-driven dynamic is desirable. That is, there are situations in which it is desirable for servers to send multiple responses without the need for user actions generating requests for each response.

In a pattern we've seen before, efforts to provide this alternative have resulted in two mechanisms—one that focuses on the server side and the other that focuses on the client side. In the server-side mechanism, called **server push**, control of the client-server transaction is maintained directly by the server. The HTTP connection is left open, and the server sends data to the client any time it wants until either the server signals the end of the transaction or the transaction is interrupted by the client. In client-side mechanism, called **client pull**, requests are generated automatically, without user action, at specified intervals, and the HTTP connection life cycle follows the pattern typical of client-server transactions initiated by user action.

As we pointed out earlier, client pull transactions follow the same HTTP connection life cycle just described, including the use of cookies to track clients and data. Client pull transactions differ from conventional user-driven client-server interactions in one important way: In conventional user-driven client-server interactions, a user directs the browser's request that the server fulfills. However, with client pull, the server or servlet (not the user) directs the browser's request. With client pull, HTTP response headers, whether generated by an application like a servlet *or* by HTML tags, direct the browser to retrieve either the same page or another page after a specified interval of time.

Now that we've described what server push and client pull are, let us say one more thing about server push before going on to ways of implement client pull. For our purposes in this course, it is enough that you be aware of what the server push mechanism is. The term is used in technical publications, and you may encounter it. However, server push will not figure into what we do later in the course. On the other hand, we will be working on client pull, both later in this page and when we come to servlets.

**Refresh**

As mentioned earlier, client pull involves directing a browser via information in an HTTP response header, information that originates either in an application or in an HTML tag. But, in order to explain these two approaches in more detail we are required to say something about the anatomy of server responses. The responses that servers send to clients usually consist of the following parts: a status line, one or more response headers, a blank line, and an HTML document. With the application-generated-response-header approach to implementing client pull, the information directing the client to retrieve a page at a specified interval is inserted into the response as a header by an application, such as a servlet—that is, this information does not come from the HTML page that is served in the response.

However, client pull can be implemented with information provided by an HTML document using the HTML **META** element. In general, the **META** element provides a means by which an author can specify information *about* an HTML document. For example, with **META** elements, authors can specify such information as a document's size, title, author, and the like. But, more relevant to our discussion about client pull is this: With the **http-equiv** attribute, the **META** element provides a means to specify information to a browser that the browser might ordinarily get from the document's HTTP response headers. **META** elements are placed within a document's **HEAD** element (that is, within its <HEAD> and </HEAD> tags), and when HTTP servers serve the document, they read its **META** elements and create HTTP response headers for all items defined by an **http-equiv** attributes and assigned values with accompanying **content** attributes.

For example, the **META** element with the **http-equiv** and **content** attributes shown below

<META http-equiv='Published' content='Sat, 22, June 2002'>

will provide a browser with the information equivalent to the following response header:

Published: Sat, 22, June 2002

That said, simple client pull can be implemented from within an HTML document with a **META** element that has an **http-equiv** attribute set to "Refresh" and a **content** attribute set to the desired number of seconds between each page reload.

<!DOCTYPE HTML PUBLIC '-//W3C//DTD HTML 4.01 Transitional//EN'>  
<HTML>  
<HEAD>

<META http-equiv='Refresh' content='5'>

<TITLE>ACME Demo Page</TITLE>

</HEAD>  
<BODY>

<H1>ACME Demo Page!</H1>

This page demonstrates an implementation of basic client pull. The page will reload every five seconds.

</BODY>  
</HTML>

**Listing 1** Simple client pull with a five second reload interval

The **META** element in Listing 1 will provide a browser with the information equivalent to the following HTTP response header

Refresh: 5

One common use of **META**-implemented client pull is to "redirect" traffic for one URL to another, say, when the first has become obsolete. In that case, the author includes in the **content** attribute the URL to which traffic is to be redirected after the interval specification and delimited from it with a semicolon (;), as shown in Listing 2.

<!DOCTYPE HTML PUBLIC '-//W3C//DTD HTML 4.01 Transitional//EN'>  
<HTML>  
<HEAD>

<META http-equiv='Refresh' content='15; URL= http://www.example.com/ACME/newACMEDemoPage.html'>

<TITLE>ACME Demo Page</TITLE>

</HEAD>  
<BODY>

<H1>ACME Demo Page!</H1>

This page demonstrates an implementation of a "redirect" using client pull. The page will redirect traffic after this page is displayed for 15 seconds.

</BODY>  
</HTML>   
**Listing 2**

The **META** element in Listing 2 will provide a browser with the information equivalent to the following HTTP response header

Refresh: 15; URL= http://www.example.com/ACME/newACMEDemoPage.html

Client pull is also used to implement "cover pages." These are pages that are displayed for a few seconds and then give way to other pages, whose URLs are specified with **content** attributes.

However, when you implement a redirect in this way, there are a couple of things you should be aware of. First, although Netscape and Microsoft browsers support **http-equiv** initiated refresh, not all browsers do. Therefore, if you use this technique to redirect traffic to another page, be sure to include a link to the new page on the page doing the redirect. Second, some browsers, if they are busy when the time for the redirect arrives, may fail to execute the redirect. For example, if they are busy, say, still loading graphics for the first page, they may not redirect traffic to the second. Therefore, it is important to set the length of the interval long enough to allow the page to load completely over the slowest connection your users might be expected to use.

**Two Client-Pull Demos**

Now, let's take a look some examples of HTML-page implemented client pull in action. [Simple Client Pull Demo](http://www.icarnegie.com/content/SSD/SSD1/2.1/normal/pg-www/pg-forms-servlets/pg-servlets-connections-refresh_v2/refresh.html" \t "externalWindow) is an example of a page that implements simple client pull. What will appear in an external browser window will resemble the image you see in Figure 7.

|  |
| --- |
|  |
| **Figure 7** *Simple Client Pull: a single page*  *being refreshed at regular intervals* |

When the Simple Client Pull Demo opens in a browser window, notice that the page reloads every five seconds. However, since the content of the page is static and does not change between each reload, there are no visible changes in what is displayed. The only thing indicating that the reloads are taking place is the status information provided by the browser. For example, in Figure 7, Internet Explorer signals the completion of a reload with the message "Done" and a solid blue status bar at the bottom of the browser window.

It should be obvious from this example that implementing simple client pull—when the content of the page remains static—accomplishes very little. However, you will see when you begin to work with servlets that some of the magic that servlets can work is to change the content of an HTML page, even as its name remains the same. Therefore, in the context of that servlet-provided ability, simple client pull can take on a range of possibilities.

Still, even with static HTML pages, it is possible to work a little magic with **META** elements—magic that goes beyond mundane applications like redirects and cover pages. For example, you can implement a simple type of animation by using the **META** element to create a series of redirects to different pages, one page redirecting to the page showing the next in a sequence of actions. To see a demonstration of this application, click [Client Pull Animation](http://www.icarnegie.com/content/SSD/SSD1/2.1/normal/pg-www/pg-forms-servlets/pg-servlets-connections-refresh_v2/a.html" \t "externalWindow). In the demonstration, random black squares converge to form the letters "iC."

**1.3.3 Using Forms with Servlets**

In our last section, we took a brief overview of servlets and the role they play on the server side of things.  In the section before that, we learned about the HTML **FORM** element and its role in providing users controls with which to input and modify data to be sent to a server, and, although we went into some detail about the **FORM** element's components, we stopped short of looking at the details of the form submission process. In this section, we focus again on forms and the client side of things and look in detail at the form submission process. Specifically, we want to know how forms send data to specific processing agents, such as servlets, and we will do this first by examining a practical example that you will be able try out using your personal server, the iCarnegie Servlet Workbench. Then, we will walk through the steps for building a simple form of your own—again, trying this with your server, the iCarnegie Servlet Workbench.

**Form Submission and Servlets**

The process of sending forms to servers and then to servlets (or other types of processing agents) consists of four steps. These are as follows:

1. All successful controls are identified.
2. A form data set is built of control name—value pairs.
3. The form data set is encoded.
4. The form data set is sent to a server addressed to a servlet or other processing agent.

This process begins with the activation of a form submit button (incidentally, a form may have more than one submit button). When this occurs, the process identifies all successful controls—that is, all controls with current values that are valid for submitting to a server. In general, these include all controls that have initial values set with their **value** attributes or have valid current values as a result of actions occurring after the form was loaded. Exceptions to this are radio buttons and checkbox controls. To be successful, radio buttons and checkbox controls must 1) be "checked" and 2) must have initial values that are assigned with **value** attributes. Unlike the values of other controls, the values that radio button and checkbox controls have, if these controls have **value** attributes, cannot be modified by user action; therefore, these controls must have default values assigned to them with **value** attributes; otherwise, they are considered "undefined" and cannot be successful controls.

Once all successful controls are identified, the browser builds the form data set. This data set consists of the name-value pairs of all successful controls. (Just how these name-value pairs are assembled is beyond the needs of this course; however, we'll get brief glimpse of some of these details in a moment when we come to the GET method.) Once this data set of name-value pairs is assembled, the set is encoded and sent to a server, addressed to a processing agent.

The way the data set is sent to the server and the agent to which it is sent are determined by a **FORM** element's **action** and **method** attributes. The **action** attribute identifies the agent (such as a servlet) by name and its location on the server, relative to the server's root directory. Therefore, this last piece of information is stated in terms of a path. The **method** attribute, on the other hand, identifies the protocol by which the data set is to be sent. This attribute can have one of two values: **get** or **post**. In general, the POST method is used for transactions that involve large amounts of data or when security is a requirement. The GET method, on the other hand, is generally used for transactions that do not involve a large amount of data and when security is not a requirement.

Incidentally, when the GET method is used, it is not uncommon for the values of the **action** attribute and the form data set to be visible in a browser's **Address** box, as seen in Figure 1. In the **Address** box in Figure 1, appended to the server's URL ("http://www.google.com") is the processing agent path and name ("/search"). Appended to the name of the processing agent is the form data set, with a question mark (?) as a delimiter. The form data set itself consists of control name-value pairs—with name and values joined by equals signs (=)—and pairs separated from each other by ampersands (&). In fact, judging from the request displayed in the **Address** box, we conclude that the form that generated the request consisted of HTML statements like the following: <form name='someName' method='get' action='/search'>, <input type=text name=q value=''>, and <input type=submit name='btnG' value='Google Search'>.

|  |
| --- |
|  |
| **Figure 1** *Servlet name and path and GET-style form*  *data set appended to a server's URL* |

**The WelcomeForm Example**

Now, let's take a close, hands-on look at a form that submits data to a servlet. The WelcomeForm solicits a user's name, which it receives in a single-line text input control. When its submit button is activated, the form submits the contents of the input control to the servlet, and the servlet in turns generates a welcome message in the form of an HTML page and returns it to the WelcomeForm's client browser. Download the files provided by the following links:

* [WelcomeForm.html](http://www.icarnegie.com/content/SSD/SSD1/2.1/normal/pg-www/pg-forms-servlets/pg-use-forms-servlets/WelcomeForm.html" \t "externalWindow) [Welcome.class](http://www.icarnegie.com/content/SSD/SSD1/2.1/normal/pg-www/pg-forms-servlets/pg-use-forms-servlets/Welcome.class)

Once they've downloaded, load them to the appropriate folders in your iCarnegie Servlet Workbench. Of course, load WelcomeForm.html to the Workbench's Content folder and Welcome.class to its Java\_Classes folder. **But, be warned:** when you are attempting this for the first time, it would be a good idea to do so under the direction of your teacher; otherwise, proceed at your own risk: although the process is relatively straightforward, there are a few pitfalls that newcomers sometimes fall into. (If you haven't done so already, you will need to download and install the iCarnegie Servlet Workbench before you can complete this step. Both a download link and instructions for installing and using the iCarnegie Servlet Workbench are available in the course appendix, see [Appendix B. The iCarnegie Servlet Workbench](javascript:ContentByName('pg-ic-servlet-workbench-appendix');).)

Now, take a moment to examine contents of WelcomeForm.html file. To do this, you might open it in a plain text editor like Notepad. When you do, what you'll see is shown in Listing 1 below:

<!DOCTYPE html public '-//W3C//DTD HTML 4.01 Transitional//EN'>  
<HTML>  
<HEAD>  
<TITLE>WelcomeForm</TITLE>  
</HEAD>

<BODY bgcolor='#fdf5e6'>  
<H1 align='center'>SSD1: Using Forms with Servlets</H1>

<FORM action='/servlet/Welcome' method='post'>  
<BR>  
<BR>  
<BR>  
<CENTER>  
Please enter your name:  
<INPUT type='text' size='20' name='Name' value=''>  
<BR>  
<BR>  
<BR>  
<INPUT type='submit' value='Done' name='userRequest'>  
</CENTER>  
</FORM>

</BODY>  
</HTML>

**Listing 1** *Source for WelcomeForm.html*

As you will notice, WelcomeForm.html is a rather simple HTML page that contains one **FORM** element. This element in turn contains HTML markup tags, some text, and two **INPUT** elements—one that creates a single-line text input control and the other, a submit button.

<FORM action='/servlet/Welcome' method='post'>  
<BR>  
<BR>  
<BR>  
<CENTER>  
Please enter your name:  
<INPUT type='text' size='20' name='Name' VALUE=''>  
<BR>  
<BR>  
<BR>  
<INPUT type='submit' value='Done' name='userRequest'>  
</CENTER>  
</FORM>  
**Listing 2** *The WelcomeForm.html file's* ***FORM*** *element*

For the moment, let's turn our attention to the **FORM** element's opening tag: <FORM action='/servlet/Welcome' method='post'>. The tag's **action** attribute specifies both the name of the servlet that is to process the form, "Welcome," and the location of that servlet on the server, in terms of an alias: "/servlet/". The alias "/servlet/" tells the server that it will find the servlet in the location designated by the alias; the actual directory assigned to this alias is determined by a setting in the server's configuration. Next, you will see that the **method** attribute specifies that the POST method be used to send the form data to the server.

Lower and centered on the form are a single-line text input control and a submit button.

...  
<BR>  
<BR>  
<BR>  
<CENTER>  
Please enter your name:  
<INPUT type='text' size='20' name='Name' VALUE=''>  
<BR>  
<BR>  
<BR>  
<INPUT type='submit' value='Done' name='userRequest'>  
...  
**Listing 3** *The WelcomeForm form controls*

The single-line text input control will appear on the form just after the text, "Please enter your name:" With a **size** attribute, this control is set to be twenty-characters wide, and with a **name** attribute, it is assigned the control name of "Name." Its **value** attribute specifies its initial value as "empty"—which is signified with two single quotation marks ('') and no intervening characters.

The submit button in turn is named "userRequest" and given an initial value of "Done". Recall that with buttons, the value specified with the **value** attribute appears as the button's label, as "Done" will be in this case. That value, "Done," will also be what is submitted with the form, paired with the control name "userRequest".

Now that we've looked at WelcomeForm.html's source, let's try the form out using the iCarnegie Servlet Workbench. If your Workbench isn't already running, start it up. Then, assuming that you've already loaded the appropriate files to the Workbench's Content and Java\_Classes folders, find WelcomeForm.html in the Content folder, and right-click it. Select the **Open in Browser** command on the shortcut menu that then appears.

|  |  |
| --- | --- |
|  |  |
| **Figure 2** *Right-click WelcomeForm.html and select the* ***Open in Browser*** *command on the shortcut menu that then appears* | **Figure 3** *A rendered WelcomeForm form* |

When WelcomeForm opens in a browser window, type a few characters in the single-line text input control—any characters will do—then click **Done**.

When the submit button is activated, the form is sent as a request to the server (your iCarnegie Servlet Workbench in this case). The server then loads and runs the servlet Welcome and passes to it the request. The Welcome servlet then generates a response—in this case, a Welcome message in the form of an HTML page—which it then sends to the server to be forwarded on to the client browser. When the browser receives this response, it renders the Welcome message.

|  |
| --- |
|  |
| **Figure 4** *A rendered Welcome message* |

**A Hands-On Example — The PersonalWelcomeForm**

Now, you try this—building a new form from scratch. In this section, we will walk step-by-step through the process and create an HTML form we'll call PersonalWelcomeForm. Like WelcomeForm, this form solicits a user's name, which it receives in a single-line text input control. It then submits the contents of the input control to a servlet when the form's submit button is activated, and the servlet in turns generates an HTML welcome message and returns it to the client browser. In fact, this form is very much like the one we've just looked at. However, the new page we will now create will be slightly different. For one, its window title and page heading will be "Personal Welcome Form" and "Servlets and Forms," respectively.

But, more important, the new page will send a form request to a different servlet: PersonalWelcome. This last difference is significant because, unlike the Welcome servlet of our last example, the PersonalWelcome servlet actually *does* something with the input we type into the single-line text input control: it incorporates this input into the welcome message that it returns to our browser. But, we'll spare you the details until later.

That said, let's get started building that new page. As we did before, begin by downloading a file—this time, the PersonalWelcome servlet—and loading it to the Workbench's Java\_Classes folder:

* [PersonalWelcome.class](http://www.icarnegie.com/content/SSD/SSD1/2.1/normal/pg-www/pg-forms-servlets/pg-use-forms-servlets/PersonalWelcome.class)

Once that is done, start creating the new page by first creating an "empty" HTML page. It might be convenient to do this in a in a plain text editor like Notepad. This "empty" HTML page consists only of a few essential elements—a **DOCTYPE** element with the appropriate attributes and both opening and closing **HTML** and **BODY** tags. When you've done this, save the file as "PersonalWelcomeForm.html."

|  |  |
| --- | --- |
|  |  |
| **Figure 5** *A rendered Personal Welcome Form* | **Figure 6** *A rendered Personal Welcome Form with a name entered* |

Note that the idea here is for you to have the opportunity to create this new page from scratch. Although, when it is finished, the new page will so closely resemble the one we looked at earlier that we could create the page by copying that one and making a few changes, there is educational value in building the new page, piece-by-piece. That said, feel free to refer back to WelcomeForm.html's the listings included earlier in this page if you need help building this new page.

You are now ready to add the tags and text necessary to give the page the desired window title and page heading. Once you've added these, you are ready to add the **FORM** element—along with the necessary text, markup tags, and form controls. And while there is no special order this must be done in, for the sake of our discussion here, focus first on adding only the basic **FORM** element—that is, both its opening and closing tags and nothing in between.

This form should use the same method used by the form in WelcomeForm.html: POST. This means that the new form's **method** attribute and value should look pretty much the same as the one we saw earlier. However, unlike that form, the new form should submit its request to a different servlet—PersonalWelcome—so this form's **action** attribute value will need to be different. Hint: when you load the PersonalWelcome servlet to the Workbench, the Workbench will put the servlet in the same location as it stored the Welcome servlet, and this fact should help you determine the correct value for this form's **action** attribute.

Once you've added the basic **FORM** element and set its **method** and **action** attributes to the appropriate values, you are ready to add the text, markup, and controls necessary for the desired function and look. In point of fact, the WelcomeForm example we looked at earlier and form we are now creating are virtually identical in these last details, and you could copy that portion of the WelcomeForm.html and use it here, and your new page would function as it should.

I say ***could*** *copy that portion of the WelcomeForm.html* here, because there are a number of details you also could change—*without* preventing your new page from functioning correctly, though the changes might affect its appearance a good deal. For example, you could set the width of the single-line text input control to a certain number of characters or include more or different markup tags and the like. Make these sorts of changes, and the PersonalWelcome servlet will still generate the correct response:

|  |  |
| --- | --- |
|  |  |
| **Figure 7** *A Personal Welcome message* | **Figure 8** *A Personal Welcome message with null for a name* |

And, while playing with these particulars could be an interesting and instructive activity, it is worth pointing out here that there are two details you must **not** change; otherwise, your new page will **not** function as it should. Your new form must have at least one single-line text input control *and* that control must be named "Name". The reason for this is that the PersonalWelcome servlet expects a control of that type—with that exact name—and if it doesn't receive one, it will respond by generating a message that contains the word "null" where the control's value should be: "Welcome to iCarnegie, null!" This is true even if the control is named "name"; that is, the PersonalWelcome servlet is case-sensitive.

Now that you've gotten to this point, there are two things you can do before loading your new page to the iCarnegie Servlet Workbench. First, you can compare the source of the PersonalWelcomeForm you've just created with the source of a working example: [PersonalWelcomeForm\_workingExample](http://www.icarnegie.com/content/SSD/SSD1/2.1/normal/pg-www/pg-forms-servlets/pg-use-forms-servlets/PersonalWelcomeForm_workingExample.html" \t "externalwindow). Next, you should validate your new page using the [WDG HTML Validator](http://htmlhelp.com/tools/validator/upload.html" \t "externalWindow). Once you've done these two things, load your new page to the Content folder of the Workbench and open it in a browser. Enter a name and then click the **Done** button, and what you see should look like Figure 7 (except the name perhaps).

**Unit 2. Introduction to Java and Object-Oriented Programming**

**2.1 Programming with Java**

With this module, the course introduces the use of objects in Java programs. It then covers the mechanics and fundamentals of Java servlet development, along with recommendations for a structured approach.

## 2.1.1 Programming with Objects

## At this point, we have a good foundation of how to design and use HTML forms from Unit 1. We are now ready to learn how to develop Java programs that can help us model many useful and interesting phenomena we commonly encounter. Some of these include making a travel reservation, purchasing books online, and building a playlist for listening to music on the Web.

Before actually writing any code in the Java programming language, we will be revisiting some of the concepts we covered in our readings of Chapter 1 of the textbook. By looking at the execution of a given Java program that models the aquatic life in a lake, we will review our understanding of the following concepts:

* A Java program is a collection of objects that communicate with each other to accomplish a common objective.
* Programs are usually written to perform a task or find a solution to a problem. Before writing a program, it is helpful to be able to model the actual process of performing a task or solving the problem. Writing the program then involves representing or specifying the model in a programming language that computers can understand.
* The building blocks or model elements in Java programs are called *objects*.
* Objects that exhibit the same behavior can be grouped into categories called *classes*.
* Objects that collaborate to perform a task or solve a problem communicate with each other by sending *messages*.

**Object-Oriented Programs**

We know from the readings that a Java program typically consists of a group of objects that communicate by sending messages. We also know that, when the program runs on a computer, it creates objects from class definitions provided by the Java programmer. A class definition for a class is a template for creating objects of that class. It describes how objects of that class must behave in terms of the types of messages they can receive and how they respond to those messages. Let us now look at these concepts using the example of corporate personnel introduced in the readings.

Writing a Java program to model a corporation in real life could be a complex task, knowing that a corporation has many different employees who perform several tasks to keep the business running. We will therefore limit ourselves to only one task and two employees involved in performing that task. The task we wish to model is that of writing a project proposal. This task involves the Vice President of a department and the Project Manager of one of the many project teams the Vice President manages.

Our first step in writing a program to model this task is to think of the objects needed by the program. The task involves two personnel, so the program will need two objects to model them: an object that models the Vice President and another that models the Project Manager. Whether these are objects of one class or two different classes will depend upon the attributes and behavior of the two objects. If the two objects have similar attributes and exhibit the same behavior, the program would need only one class definition to create the two objects. On the other hand, if the two objects have different attributes and exhibit different behavior, the program would need two class definitions to create the two objects.

This leads us to consider the attributes and behavior of the two objects. While both personnel are employees of the same corporation, they have different areas of responsibility and must perform different tasks. They also have different attributes that depend upon their role at the corporation. For instance, a Project Manager can be asked to prepare a project proposal by a Vice President. Vice Presidents, on the other hand do not prepare project proposals themselves. Further, a Vice President can be asked to provide financial data pertaining to the performance of the company. Also, while a Vice President heads several project teams, a Project Manager heads only one project team. Thus, it is clear that the two objects we need in our program have different attributes and exhibit different behavior. This indicates that they belong to two distinct categories or classes. So, the program will need two classes to model them, and hence two class definitions to create the objects.

Let us now explore the two class definitions needed by the program by examining the attributes and behavior of the Vice President and Project Manager objects. Note that the behavior of the objects is tied to the type of messages they must respond to. For instance, if the task we are modeling needs the Vice President to provide financial data, then the Vice President object must respond to a message for providing financial data. Note also that an object may have several attributes, but we will consider only those relevant to modeling the task in our program. For instance, a Vice President may be the chairperson of a professional association, but this attribute of the Vice President is of no interest to us in modeling the task for writing a project proposal.

Details for the sequence of steps in the task to be modeled by our program are as follows:

* The Vice President requests the Project Manager to write a proposal for a project. In response to this message, the Project Manager starts writing the proposal and realizes that financial data is needed from the Vice President.
* The Project Manager asks the Vice President for financial data. In response to this message, the Vice President sends the financial data to the Project Manager.
* The Project Manager completes the proposal with the financial data and other necessary information.

Based on the above interaction, the Vice President and Project Manager classes must model the following attributes:

* Attributes for the Vice President object:
  + Name
  + Department
* Attributes for the Project Manager object:
  + Name
  + Team

It is easy to see that the attributes of the Vice President object identify the particular department. Similarly, the attributes of the Project Manager object identify the project team.

Further, the above interaction requires that the Vice President and Project Manager objects respond to messages as follows:

* Message(s) from the Vice President object that the Project Manager object must respond to :
  + Write project proposal
* Message(s) from the Project Manager object that the Vice President object must respond to:
  + Provide financial data

The classes that model the Vice President and Project Manager must therefore describe how the respective objects would behave in response to these messages.

We now have some information about the attributes and behavior for the Vice President and Project Manager objects. In other words, we have the necessary information for the class definitions that could be used by our program to create these objects. The following diagram is a simplified representation of these classes:



**Fish Simulation: An OO Program that Simulates Aquatic Life**

Let us now extend our knowledge of objects and object-oriented programs to a simulation program written in Java. By observing the execution of the program, we will attempt to understand how this program may have been built, in particular the various objects used by the program and the communication between those objects.

As you may already know, a simulation is really a model of a system, process, or environment. Simulations allow us to assess and understand the system, process, or environment being modeled by conducting experiments with the different elements of the model. For instance, an aeronautical engineer might build a working model of a new aircraft being designed. The model aircraft simulates a real aircraft. The simulation offers the engineer a more convenient and comparatively inexpensive method to make observations and predictions about the behavior of the real aircraft by conducting experiments on the smaller and more manageable model aircraft. Simulation programs are an extension of this concept. They allow us to harness the power of computers for simulation. Typically, simulation programs are executed as a series of time steps. The simulation program we will examine is one such program that simulates the life and movements of organisms such as fish and algae in a lake.

To run the simulation program on your machine:

* Download and extract the contents of the zip file [Alife.zip](http://www.icarnegie.com/content/SSD/SSD1/2.1/normal/pg-intro-java-oop/pg-prog-java/pg-objects-java/Alife.zip) to your machine.
* In the iCarnegie Servlet Workbench window, click Load Content on the Actions menu, and load the *.html* and *.gif* files extracted from the zip file.
* In the iCarnegie Servlet Workbench window, click Load Java on the Actions menu, and load the *.class* files extracted from the zip file.
* Select the file *initialWorldAlgaeFishCroc.html* in the workbench window, and open it with the Open in Browsercommand on the Actions menu.

Notice that the file *initialWorldAlgaeFishCroc.html* contains an HTML form with a grid of 10 rows with 10 cells per row. Each cell of the grid contains three checkbox input controls labeled *algae*, *catfish*, and *crocodile*. The form also contains a textbox input control below the grid, labeled *Blocks of Time to Simulate*. At the end of the form is a button labeled *Start Simulation*.

The grid in the form essentially models the body of water in the lake. For now, the organisms being modeled include catfish, crocodiles, and algae. Accordingly, each cell of the grid allows a choice of any combination of these three organisms. One of the goals of the simulation is to study how the movement of catfish and crocodiles and the growth of algae are affected by the size and density of the populations. To do this, we plan to run several iterations of the simulation program. Further, for each iteration, we plan to set up the initial condition of the lake by varying one or more of the following: the number and starting location of catfish, crocodiles, and algae.

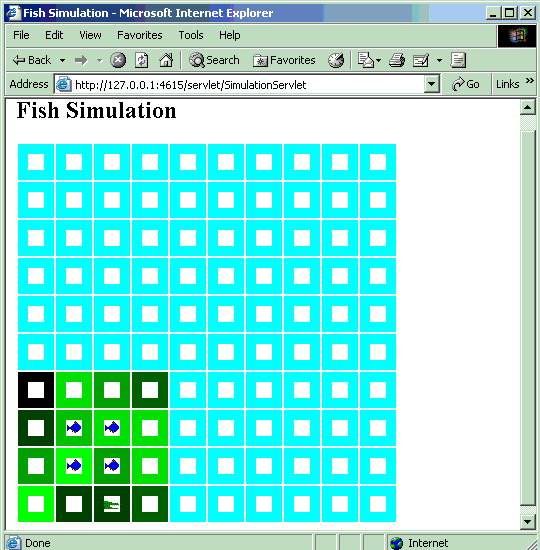
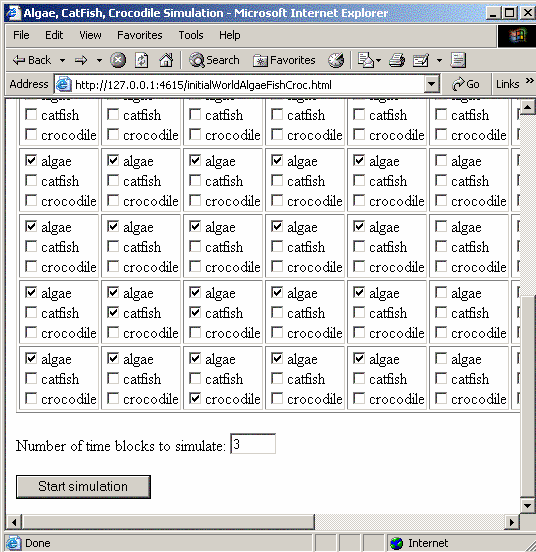
Although we will observe the behavior of the organisms when the simulation program runs, here is some information to help you set up the initial condition of the lake:

* Catfish are capable of eating algae, and crocodile are capable of eating catfish.
* Both catfish and crocodiles can move from cell to cell, while algae are stationary.
* Algae require sunlight to survive. Random amounts of sunlight per cell are provided by the program.
* Typically, a choice of 20–30 time steps is enough to observe the behavior of the organisms and draw some conclusions.

Here are a few initial conditions for you to try out:

* One catfish, no algae, and no crocodile
* One crocodile, no algae, and no catfish
* Algae only, no catfish and no crocodile
* A colony of algae, with a couple of catfish and a crocodile at a distance from the algae colony

The following screenshot shows the form to set up the initial configuration of the lake:



The following screenshot shows what your browser may look like with the last configuration:

After you have done some experiments on the model, see if you can answer the following questions by drawing on your knowledge of object-oriented programs:

1. What classes would be required to build such a model that simulates the life of organisms in a lake?
2. Does the simulation program always need to create objects of these classes? In other words, how does your initial choice of configuration affect the creation of objects by the program?
3. How does the number of organisms selected by you in the initial configuration of the lake affect the number of objects created by the program?
4. What would be the attributes of the classes you thought of as answer to #1? Hint: you need to observe the appearance and behavior of the organism being modeled by the class to deduce the attributes. In addition, you need to take into consideration any information you were given at the start. For instance: knowing that algae need energy to live and having observed that algae sometimes die in the simulation tells us that the class that models algae needs an attribute for the minimum amount of energy needed to stay alive.
5. Does the crocodile's behavior change based on the presence or absence of catfish; does the crocodile move toward the fish or away from the fish; and do these questions affect how the crocodile should be modelled? Similarly, does the catfish's behavior change based on the presence or absence of algae; does the catfish move toward the algae or away from the algae; and do these questions affect how the catfish should be modelled?

Note that you will need to run the simulation several times to be able to answer these questions. It is also acceptable if you feel that you do not have enough information at this point in the course to answer a particular question. When you have attempted them all, you will realize that the questions collectively test your understanding of the relationship between classes and objects, the structure of an object, and the organization of object-oriented programs around objects that communicate using messages.

As you progress through the rest of the course, we will look at more details about how this simulation program is built in Java. In fact, you may actually build an extension to it using the concepts you will learn in the course, such as modeling additional organisms in the lake. To be able to do this, we will next look at how Java programs are written and prepared for execution.

**2.1.2 Java Program Development**

As you have seen in [1.3 Introduction to HTML Forms and Servlets](javascript:ContentByName('pg-forms-servlets')), servlets are Java programs that extend the functionality of a Web server. So far, we have learned how to invoke servlets using forms and the iCarnegie Servlet Workbench, but we do not know anything about how servlets are created. We have also examined a simulation program that used objects to provide interaction between a servlet and a form.

At this point, you should have gained some knowledge about writing, compiling and executing Java programs from the readings in Chapter 1 of the textbook. We will now use that knowledge to develop a Java servlet, which is a type of Java program, using the same "Welcome"servlet that was invoked by the form used in [1.3.3 Using Forms with Servlets](javascript:ContentByName('pg-use-forms-servlets')). We will also develop a Java class, using the class definition for one of the objects, *Catfish*, used by the simulation program in [2.1.1 Programming with Objects](javascript:ContentByName('pg-objects-java')). Note that we will be working with a simplified version of the simulation program at this time. So, before you get started on the tasks in this section, please delete all files that were loaded into the iCarnegie Servlet Workbench for the simulation program in [2.1.1 Programming with Objects](javascript:ContentByName('pg-objects-java')).

Overall, in this section, we will be re-visiting the following concepts covered earlier in the course and in the readings:

* As seen in [1.3.3 Using Forms with Servlets](javascript:ContentByName('pg-use-forms-servlets')), to invoke the"Welcome" servlet, the **action** attribute of the **FORM** element in the form must be set to the file *Welcome.class.*
* As covered in section 1.6 of the textbook, source code contained in *.java* files can be compiled to produce Java Byte Code in *.class* files. For instance, the file *Welcome.class* contains Java Byte Code. It is the result of compiling the Java program file *Welcome.java*. In this section:
  + We will look at how to create the file *Welcome.java* and how to compile it to obtain the file *Welcome.class.*
  + We will also look at how to create the file *Catfish.java* that defines the attributes and behavior for the *Catfish* objects used by the simulation program we worked with in [2.1.1 Programming with Objects](javascript:ContentByName('pg-objects-java')). Note that we will be working with a simplified version of the simulation program in this section, since we are yet to learn Java in detail.
  + We will further look at how to compile *Catfish.java* to obtain the file *Catfish.class*.
* As covered in section 1.7 of the textbook, a Java Virtual Machine is required for executing a Java program. As seen in [1.3.3 Using Forms with Servlets](javascript:ContentByName('pg-use-forms-servlets')), when the *Welcome.class* file is executed using the iCarnegie Servlet Workbench, the "Welcome"servlet would execute on the JVM that is installed on the Web server included in the workbench.

To develop a program in the Java programming language, you will need the Java 2 Platform Standard Edition (J2SE) development kit. If you have successfully worked through the example in the textbook for writing, compiling, and executing a Java program, you will already have this installed on your machine. If not, please refer to [Appendix A. Java 2 Platform, Standard Edition (J2SE)](javascript:ContentByName('pg-sdk')) for instructions on downloading and installing this kit.

**Writing Java Source Code**

Just like the example in 1.7 of the textbook, our first step involves starting an editor program, typing in the Java program and saving the contents to a file called *Welcome.java.* Hereafter, we will refer to such files containing source code as *source* files, or *source-code* files. In particular, we may also refer to source files for a servlet as *servlet source* files.

Listing 1 shows the source code for the servlet *Welcome*. At this time, type or copy-and-paste the following source code into an editior, such as Notepad. Then, save the file. Remember that the source file must be named *Welcome.java* since the name of the servlet is *Welcome*, as seen on line 5 of the listing.

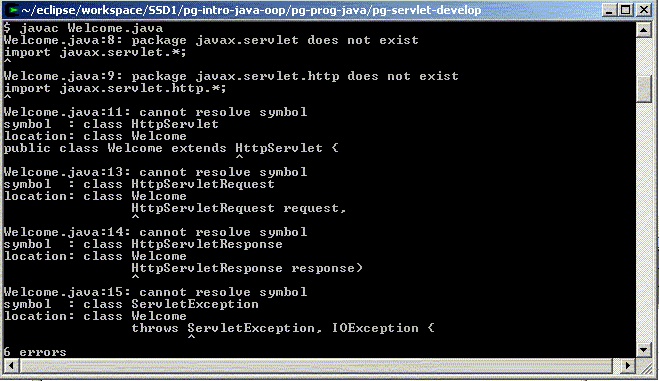
|  |  |
| --- | --- |
| 1:  2:  3:  4:  5:  6:  7:  8:  9: 10: 11: 12: 13: 14: 15: 16: 17: 18: 19: 20: 21: 22: 23: 24: 25: 26: 27: | import java.io.\*;  import javax.servlet.\*;  import javax.servlet.http.\*;  public class Welcome extends HttpServlet {  public void doPost(HttpServletRequest request,  HttpServletResponse response)  throws ServletException, IOException {  /\*\*  \* Indicate the content type (for example, text/html),  \* being returned by the response  \*/  response.setContentType("text/plain");  /\*\*  \* Retrieve an output stream to use to send data  \* to the client  \*/  PrintWriter out = response.getWriter();  /\*\*  \* Write the response  \*/  out.println("Welcome to iCarnegie!");  }  } |
| **Listing 1** *Welcome.java* | |

When creating the source files, please pay close attention to the program format and comments as noted in the Java Interludetitled "Identifiers, Statement Order, Format, and Comments"of the textbook. Please note that the line numbers in the listing are not part of the source code and should not be typed in.

**Compiling Java Source Code**

As the next step, compile the source files using the *javac* command at the system prompt to invoke the Java compiler that is included in the Java 2 Standard Edition, Software Development Kit (J2SE SDK).

Note that, while the J2SE SDK is sufficient to compile most Java programs, we need something additional to compile a Java servlet. This is because the source code for the servlet uses some pre-written Java source code that is not part of the J2SE SDK. This code is included in the file *javax.servlet.jar* located in the iCarnegie Servlet Workbench installation in the *lib* directory. When compiling servlets, the compiler locates this *.jar* file by examining the system's **classpath** environment variable. Thus, for successfully compiling servlets, the file *javax.servlet.jar* *must* be added to the **classpath**. The following screenshot shows the compile-time error message you will see if you miss this important step.



The [Appendix B. The iCarnegie Servlet Workbench](javascript:ContentByName('pg-ic-servlet-workbench-appendix');) has more information on how to edit your system's **classpath** environment variable. Based on the instructions in the Appendix B of the textbook and [Appendix A. Java 2 Platform, Standard Edition (J2SE)](javascript:ContentByName('pg-sdk');), compile the servlet source file using the following command at the system prompt: javac Welcome.java

**Creating an HTML Form**

Unlike the example in 1.7 of the textbook, we need to take an additional step in preparing our program for execution. This is because we need an HTML form to invoke the "Welcome" servlet, as seen in [1.3 Introduction to HTML Forms and Servlets](javascript:ContentByName('pg-forms-servlets')). A sample Web page containing a form that invokes the "Welcome" servlet follows:

<!DOCTYPE HTML PUBLIC '-//W3C//DTD HTML 4.01 Transitional//EN'>

<HTML>

<HEAD>

<TITLE>WelcomeForm</TITLE>

</HEAD>

<BODY bgcolor='#fdf5e6'>

<H1 ALIGN="center">Forms and Servlets</H1>

<FORM action='/servlet/Welcome' method='post'>

<BR>

<BR>

<BR>

<CENTER>

Please enter your name:

<INPUT type='text' size='20' name='Name' value=''>

<BR>

<BR>

<BR>

<INPUT type='submit' value='Done' name='userRequest'>

</CENTER>

</FORM>

</BODY>

</HTML>

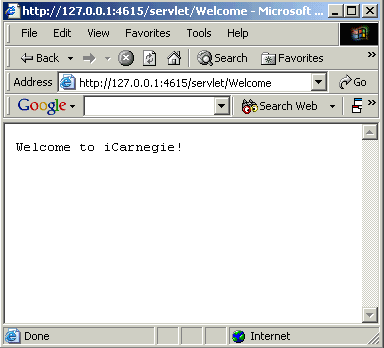
As shown in [1.3 Introduction to HTML Forms and Servlets](javascript:ContentByName('pg-forms-servlets')), create the file *WelcomeForm.html* with the HTML code shown above. We will use this file to run the servlet. Note that the **action** attribute of the **FORM** element in the HTML code refers to the "Welcome"servlet class file as shown below, even though the file extension *.class* is not mentioned.

<FORM action='/servlet/Welcome' method='post'>

**Running a Java Program**

Now that we have the servlet class file *Welcome.class* and a form that invokes it in *WelcomeForm.html*, we are ready to execute the servlet using the iCarnegie Servlet Workbench. As in [1.3 Introduction to HTML Forms and Servlets](javascript:ContentByName('pg-forms-servlets')), first load the file *WelcomeForm.html* with the Load HTMLcommand on the Actionsmenu of the workbench. Next, load the file *Welcome.class* with the Load Javacommand on the Actionsmenu of the workbench.

Finally, select the file *WelcomeForm.html* from the *Content* directory within the workbench, and then select the Open in Browsercommand on the Actionsmenu. The following screenshot shows the form before it is submitted.



The "Welcome" servlet is invoked when the form is submitted, and the browser displays the welcome message sent by the servlet, as follows:

Having learned how to develop a Java program, see if you can apply your knowledge to developing the *Catfish* object used by a simplified version of the simulation program from [2.1.1 Programming with Objects](javascript:ContentByName('pg-objects-java');). Listing 2 shows the source code for the class *Catfish*.

|  |  |
| --- | --- |
| 1:   2:   3:   4:   5:   6:   7:   8:   9:  10:  11:  12:  13:  14:  15:  16:  17:  18:  19:  20:  21:  22:  23:  24:  25:  26:  27:  28:  29:  30:  31:  32:  33:  34:  35:  36:  37:  38:  39:  40:  41:  42:  43:  44:  45:  46:  47:  48:  49:  50:  51:  52:  53:  54:  55:  56:  57:  58:  59:  60:  61:  62:  63:  64:  65:  66:  67:  68:  69:  70:  71:  72:  73:  74:  75:  76:  77:  78:  79:  80:  81:  82:  83:  84:  85:  86:  87:  88:  89:  90:  91:  92:  93:  94:  95:  96:  97:  98:  99: 100: 101: 102: 103: 104: 105: 106: 107: 108: 109: 110: 111: 112: 113: 114: 115: | public class Catfish {  /\*\*  \* Location of the catfish - which row.  \*/  private int row;  /\*\*  \* Location of catfish - which column  \*/  private int column;  /\*\*  \* Image of the catfish - is really a filename.  \*/  private String imageFileName;  /\*\*  \* Location of catfish - the row  \*  \* @return - an integer representing the  \* row location of catfish.  \*/  public int getRow() {  return row;  }  /\*\*  \* Location of catfish - the column  \*  \* @return - an integer representing the  \* column location of catfish.  \*/  public int getColumn() {  return column;  }  /\*\*  \* Swim one cell to the right.  \* The swimming area will be limited to the lake boundary.  \* If catfish is at the right boundary, it will not  \* swim further right.  \*  \* @return nothing.  \*/  public void swimRightIfPossible() {  // If the cell to the right is within bounds, move right.  if (column + 1 <= 10) {  column = column + 1;  }  imageFileName = "/Catfish-right.gif";  }  /\*\*  \* Swim one cell to the left.  \* The swimming area will be limited to the lake boundary.  \* If catfish is at the left boundary, it will not  \* swim further left.  \*  \* @return nothing.  \*/  public void swimLeftIfPossible() {  // If the cell to the left is within bounds, move left.  if (column - 1 >= 1) {  column = column - 1;  }  imageFileName = "/Catfish-left.gif";  }  /\*\*  \* Swim one cell down.  \* The swimming area will be limited to the lake boundary.  \* If catfish is at the bottom boundary, it will not  \* swim further down.  \*  \* @return nothing.  \*/  public void swimDownIfPossible() {  // If the cell below is within bounds, move down.  if (row + 1 <= 10) {  row = row + 1;  }  imageFileName = "/Catfish-down.gif";  }  /\*\*  \* Swim one cell up.  \* The swimming area will be limited to the lake boundary.  \* If catfish is at the top boundary, it will not  \* swim further up.  \*  \* @return nothing.  \*/  public void swimUpIfPossible() {  // If the cell above is within bounds, move up.  if (row - 1 >= 1) {  row = row - 1;  }  imageFileName = "/Catfish-up.gif";  }  /\*\*  \* get filename of catfish image  \*  \* @return filename of Catfish image  \*/  public String getImage() {  return imageFileName;  }  } |
| **Listing 2** *Catfish.java* | |

For now, type in or cut-and-paste this source code using an editor*,* and save the contents to a file. Remember that the source file must be named *Catfish.java*, since the name of the class is *Catfish*, as seen on line 1 of the listing.

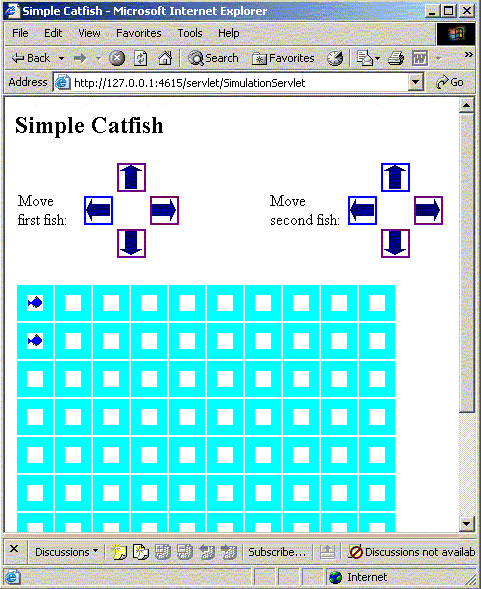
Next, we compile the source file for the class *Catfish*:

javac Catfish.java

Note that this source file does not use any pre-defined Java classes.

Now, we must prepare the *Catfish.class* file for execution. We know that this is only one of several class files used by the simulation program. At this point, we are not ready to examine all the other class files. But, we do know that the simulation program involves interaction between a servlet and a form.

If you haven't already done so, please delete the files loaded for the simulation program in [2.1.1 Programming with Objects](javascript:ContentByName('pg-objects-java');) since we are now working with a different version of the simulation program. Next, download [Alife.zip](http://www.icarnegie.com/content/SSD/SSD1/2.1/normal/pg-intro-java-oop/pg-prog-java/pg-servlet-develop_v2/Alife-basic.zip) and extract the contents to the same directory where you have been working so far. Then, load the various *.gif* files and the form in *CreateFishes.html* into the iCarnegie Servlet Workbench. Next, load the various *.class* files and the file *Catfish.class* created above into the workbench. Finally, select the file *CreateFishes.html* from the *Content* directory within the workbench, and then click Open in Browseron the Actionsmenu. The following screenshot shows what the browser window may look like when you click the button labeled *Create Two Fishes* in the form in *CreateFishes.html*:



At this point, you have successfully completed development of a Java servlet and a Java class. You know how to create a *.java* source-code file for a Java servlet and the class definition for a Java object. You also know how to compile a Java source file to get a *.class* file. Finally, you know how to run a servlet using an HTML form and the iCarnegie Servlet Workbench. In the two sections that follow, we will examine in some detail the source files we wrote in this section.

**2.1.2 Java Program Development**

As you have seen in [1.3 Introduction to HTML Forms and Servlets](javascript:ContentByName('pg-forms-servlets')), servlets are Java programs that extend the functionality of a Web server. So far, we have learned how to invoke servlets using forms and the iCarnegie Servlet Workbench, but we do not know anything about how servlets are created. We have also examined a simulation program that used objects to provide interaction between a servlet and a form.

At this point, you should have gained some knowledge about writing, compiling and executing Java programs from the readings in Chapter 1 of the textbook. We will now use that knowledge to develop a Java servlet, which is a type of Java program, using the same "Welcome"servlet that was invoked by the form used in [1.3.3 Using Forms with Servlets](javascript:ContentByName('pg-use-forms-servlets')). We will also develop a Java class, using the class definition for one of the objects, *Catfish*, used by the simulation program in [2.1.1 Programming with Objects](javascript:ContentByName('pg-objects-java')). Note that we will be working with a simplified version of the simulation program at this time. So, before you get started on the tasks in this section, please delete all files that were loaded into the iCarnegie Servlet Workbench for the simulation program in [2.1.1 Programming with Objects](javascript:ContentByName('pg-objects-java')).

Overall, in this section, we will be re-visiting the following concepts covered earlier in the course and in the readings:

* As seen in [1.3.3 Using Forms with Servlets](javascript:ContentByName('pg-use-forms-servlets')), to invoke the"Welcome" servlet, the **action** attribute of the **FORM** element in the form must be set to the file *Welcome.class.*
* As covered in section 1.6 of the textbook, source code contained in *.java* files can be compiled to produce Java Byte Code in *.class* files. For instance, the file *Welcome.class* contains Java Byte Code. It is the result of compiling the Java program file *Welcome.java*. In this section:
  + We will look at how to create the file *Welcome.java* and how to compile it to obtain the file *Welcome.class.*
  + We will also look at how to create the file *Catfish.java* that defines the attributes and behavior for the *Catfish* objects used by the simulation program we worked with in [2.1.1 Programming with Objects](javascript:ContentByName('pg-objects-java')). Note that we will be working with a simplified version of the simulation program in this section, since we are yet to learn Java in detail.
  + We will further look at how to compile *Catfish.java* to obtain the file *Catfish.class*.
* As covered in section 1.7 of the textbook, a Java Virtual Machine is required for executing a Java program. As seen in [1.3.3 Using Forms with Servlets](javascript:ContentByName('pg-use-forms-servlets')), when the *Welcome.class* file is executed using the iCarnegie Servlet Workbench, the "Welcome"servlet would execute on the JVM that is installed on the Web server included in the workbench.

To develop a program in the Java programming language, you will need the Java 2 Platform Standard Edition (J2SE) development kit. If you have successfully worked through the example in the textbook for writing, compiling, and executing a Java program, you will already have this installed on your machine. If not, please refer to [Appendix A. Java 2 Platform, Standard Edition (J2SE)](javascript:ContentByName('pg-sdk')) for instructions on downloading and installing this kit.

**Writing Java Source Code**

Just like the example in 1.7 of the textbook, our first step involves starting an editor program, typing in the Java program and saving the contents to a file called *Welcome.java.* Hereafter, we will refer to such files containing source code as *source* files, or *source-code* files. In particular, we may also refer to source files for a servlet as *servlet source* files.

Listing 1 shows the source code for the servlet *Welcome*. At this time, type or copy-and-paste the following source code into an editior, such as Notepad. Then, save the file. Remember that the source file must be named *Welcome.java* since the name of the servlet is *Welcome*, as seen on line 5 of the listing.

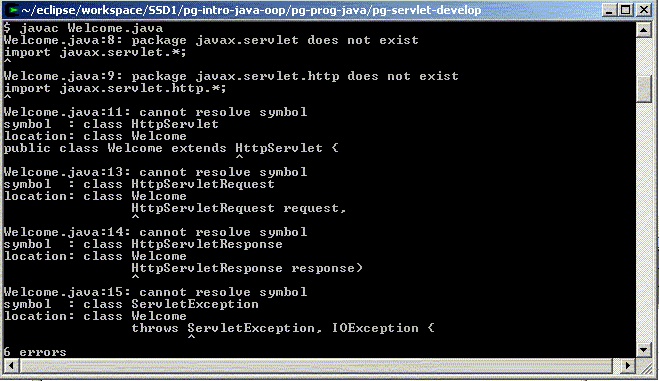
|  |  |
| --- | --- |
| 1:  2:  3:  4:  5:  6:  7:  8:  9: 10: 11: 12: 13: 14: 15: 16: 17: 18: 19: 20: 21: 22: 23: 24: 25: 26: 27: | import java.io.\*;  import javax.servlet.\*;  import javax.servlet.http.\*;  public class Welcome extends HttpServlet {  public void doPost(HttpServletRequest request,  HttpServletResponse response)  throws ServletException, IOException {  /\*\*  \* Indicate the content type (for example, text/html),  \* being returned by the response  \*/  response.setContentType("text/plain");  /\*\*  \* Retrieve an output stream to use to send data  \* to the client  \*/  PrintWriter out = response.getWriter();  /\*\*  \* Write the response  \*/  out.println("Welcome to iCarnegie!");  }  } |
| **Listing 1** *Welcome.java* | |

When creating the source files, please pay close attention to the program format and comments as noted in the Java Interludetitled "Identifiers, Statement Order, Format, and Comments"of the textbook. Please note that the line numbers in the listing are not part of the source code and should not be typed in.

**Compiling Java Source Code**

As the next step, compile the source files using the *javac* command at the system prompt to invoke the Java compiler that is included in the Java 2 Standard Edition, Software Development Kit (J2SE SDK).

Note that, while the J2SE SDK is sufficient to compile most Java programs, we need something additional to compile a Java servlet. This is because the source code for the servlet uses some pre-written Java source code that is not part of the J2SE SDK. This code is included in the file *javax.servlet.jar* located in the iCarnegie Servlet Workbench installation in the *lib* directory. When compiling servlets, the compiler locates this *.jar* file by examining the system's **classpath** environment variable. Thus, for successfully compiling servlets, the file *javax.servlet.jar* *must* be added to the **classpath**. The following screenshot shows the compile-time error message you will see if you miss this important step.



The [Appendix B. The iCarnegie Servlet Workbench](javascript:ContentByName('pg-ic-servlet-workbench-appendix');) has more information on how to edit your system's **classpath** environment variable. Based on the instructions in the Appendix B of the textbook and [Appendix A. Java 2 Platform, Standard Edition (J2SE)](javascript:ContentByName('pg-sdk');), compile the servlet source file using the following command at the system prompt:

javac Welcome.java

**Creating an HTML Form**

Unlike the example in 1.7 of the textbook, we need to take an additional step in preparing our program for execution. This is because we need an HTML form to invoke the "Welcome" servlet, as seen in [1.3 Introduction to HTML Forms and Servlets](javascript:ContentByName('pg-forms-servlets')). A sample Web page containing a form that invokes the "Welcome" servlet follows:

<!DOCTYPE HTML PUBLIC '-//W3C//DTD HTML 4.01 Transitional//EN'>

<HTML>

<HEAD>

<TITLE>WelcomeForm</TITLE>

</HEAD>

<BODY bgcolor='#fdf5e6'>

<H1 ALIGN="center">Forms and Servlets</H1>

<FORM action='/servlet/Welcome' method='post'>

<BR>

<BR>

<BR>

<CENTER>

Please enter your name:

<INPUT type='text' size='20' name='Name' value=''>

<BR>

<BR>

<BR>

<INPUT type='submit' value='Done' name='userRequest'>

</CENTER>

</FORM>

</BODY>

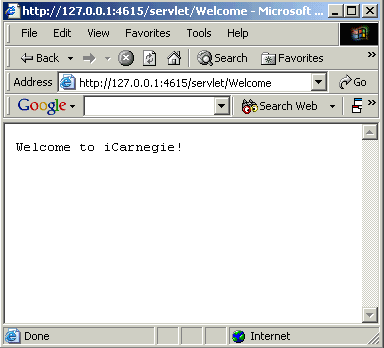
</HTML>

As shown in [1.3 Introduction to HTML Forms and Servlets](javascript:ContentByName('pg-forms-servlets')), create the file *WelcomeForm.html* with the HTML code shown above. We will use this file to run the servlet. Note that the **action** attribute of the **FORM** element in the HTML code refers to the "Welcome"servlet class file as shown below, even though the file extension *.class* is not mentioned.

<FORM action='/servlet/Welcome' method='post'>

**Running a Java Program**

Now that we have the servlet class file *Welcome.class* and a form that invokes it in *WelcomeForm.html*, we are ready to execute the servlet using the iCarnegie Servlet Workbench. As in [1.3 Introduction to HTML Forms and Servlets](javascript:ContentByName('pg-forms-servlets')), first load the file *WelcomeForm.html* with the Load HTMLcommand on the Actionsmenu of the workbench. Next, load the file *Welcome.class* with the Load Javacommand on the Actionsmenu of the workbench. Finally, select the file *WelcomeForm.html* from the *Content* directory within the workbench, and then select the Open in Browsercommand on the Actionsmenu. The following screenshot shows the form before it is submitted.



The "Welcome" servlet is invoked when the form is submitted, and the browser displays the welcome message sent by the servlet, as follows:

Having learned how to develop a Java program, see if you can apply your knowledge to developing the *Catfish* object used by a simplified version of the simulation program from [2.1.1 Programming with Objects](javascript:ContentByName('pg-objects-java');).

Listing 2 shows the source code for the class *Catfish*.

|  |  |
| --- | --- |
| 1:   2:   3:   4:   5:   6:   7:   8:   9:  10:  11:  12:  13:  14:  15:  16:  17:  18:  19:  20:  21:  22:  23:  24:  25:  26:  27:  28:  29:  30:  31:  32:  33:  34:  35:  36:  37:  38:  39:  40:  41:  42:  43:  44:  45:  46:  47:  48:  49:  50:  51:  52:  53:  54:  55:  56:  57:  58:  59:  60:  61:  62:  63:  64:  65:  66:  67:  68:  69:  70:  71:  72:  73:  74:  75:  76:  77:  78:  79:  80:  81:  82:  83:  84:  85:  86:  87:  88:  89:  90:  91:  92:  93:  94:  95:  96:  97:  98:  99: 100: 101: 102: 103: 104: 105: 106: 107: 108: 109: 110: 111: 112: 113: 114: 115: | public class Catfish {  /\*\*  \* Location of the catfish - which row.  \*/  private int row;  /\*\*  \* Location of catfish - which column  \*/  private int column;  /\*\*  \* Image of the catfish - is really a filename.  \*/  private String imageFileName;  /\*\*  \* Location of catfish - the row  \*  \* @return - an integer representing the  \* row location of catfish.  \*/  public int getRow() {  return row;  }  /\*\*  \* Location of catfish - the column  \*  \* @return - an integer representing the  \* column location of catfish.  \*/  public int getColumn() {  return column;  }  /\*\*  \* Swim one cell to the right.  \* The swimming area will be limited to the lake boundary.  \* If catfish is at the right boundary, it will not  \* swim further right.  \*  \* @return nothing.  \*/  public void swimRightIfPossible() {  // If the cell to the right is within bounds, move right.  if (column + 1 <= 10) {  column = column + 1;  }  imageFileName = "/Catfish-right.gif";  }  /\*\*  \* Swim one cell to the left.  \* The swimming area will be limited to the lake boundary.  \* If catfish is at the left boundary, it will not  \* swim further left.  \*  \* @return nothing.  \*/  public void swimLeftIfPossible() {  // If the cell to the left is within bounds, move left.  if (column - 1 >= 1) {  column = column - 1;  }  imageFileName = "/Catfish-left.gif";  }  /\*\*  \* Swim one cell down.  \* The swimming area will be limited to the lake boundary.  \* If catfish is at the bottom boundary, it will not  \* swim further down.  \*  \* @return nothing.  \*/  public void swimDownIfPossible() {  // If the cell below is within bounds, move down.  if (row + 1 <= 10) {  row = row + 1;  }  imageFileName = "/Catfish-down.gif";  }  /\*\*  \* Swim one cell up.  \* The swimming area will be limited to the lake boundary.  \* If catfish is at the top boundary, it will not  \* swim further up.  \*  \* @return nothing.  \*/  public void swimUpIfPossible() {  // If the cell above is within bounds, move up.  if (row - 1 >= 1) {  row = row - 1;  }  imageFileName = "/Catfish-up.gif";  }  /\*\*  \* get filename of catfish image  \*  \* @return filename of Catfish image  \*/  public String getImage() {  return imageFileName;  }  } |
| **Listing 2** *Catfish.java* | |

For now, type in or cut-and-paste this source code using an editor*,* and save the contents to a file. Remember that the source file must be named *Catfish.java*, since the name of the class is *Catfish*, as seen on line 1 of the listing.

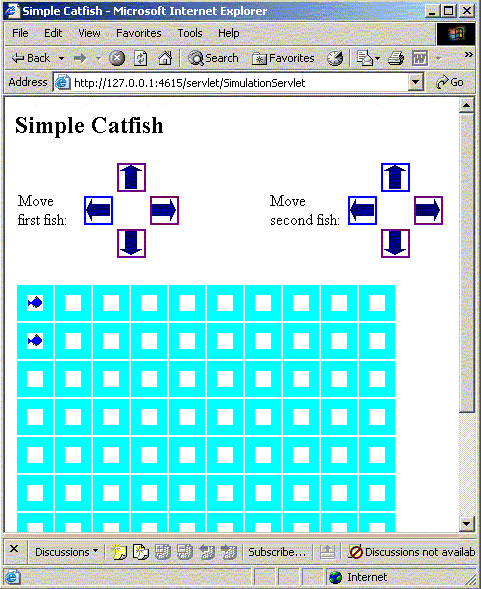
Next, we compile the source file for the class *Catfish*:

javac Catfish.java

Note that this source file does not use any pre-defined Java classes.

Now, we must prepare the *Catfish.class* file for execution. We know that this is only one of several class files used by the simulation program. At this point, we are not ready to examine all the other class files. But, we do know that the simulation program involves interaction between a servlet and a form.

If you haven't already done so, please delete the files loaded for the simulation program in [2.1.1 Programming with Objects](javascript:ContentByName('pg-objects-java');) since we are now working with a different version of the simulation program. Next, download [Alife.zip](http://www.icarnegie.com/content/SSD/SSD1/2.1/normal/pg-intro-java-oop/pg-prog-java/pg-servlet-develop_v2/Alife-basic.zip) and extract the contents to the same directory where you have been working so far. Then, load the various *.gif* files and the form in *CreateFishes.html* into the iCarnegie Servlet Workbench. Next, load the various *.class* files and the file *Catfish.class* created above into the workbench. Finally, select the file *CreateFishes.html* from the *Content* directory within the workbench, and then click Open in Browseron the Actionsmenu. The following screenshot shows what the browser window may look like when you click the button labeled *Create Two Fishes* in the form in *CreateFishes.html*:



At this point, you have successfully completed development of a Java servlet and a Java class. You know how to create a *.java* source-code file for a Java servlet and the class definition for a Java object. You also know how to compile a Java source file to get a *.class* file. Finally, you know how to run a servlet using an HTML form and the iCarnegie Servlet Workbench. In the two sections that follow, we will examine in some detail the source files we wrote in this section.

**2.1.3 First Look at Java**

We know from [2.1.2 Java Program Development](javascript:ContentByName('pg-servlet-develop')) how source code for a class definition in Java is written and prepared for execution. In this section, we will apply the concepts covered in the readings in Chapter 1 of the textbook to examine the source code for the same *Catfish* class in some detail.

**The *Catfish* Class**

The source code for the *Catfish* class from the source file in [2.1.2 Java Program Development](javascript:ContentByName('pg-servlet-develop')) is shown in Listing 1.

|  |  |
| --- | --- |
| 1:   2:   3:   4:   5:   6:   7:   8:   9:   10:  11:  12:  13:  14:  15:  16:  17:  18:  19:  20:  21:  22:  23:  24:  25:  26:  27:  28:  29:  30:  31:  32:  33:  34:  35:  36:  37:  38:  39:  40:  41:  42:  43:  44:  45:  46:  47:  48:  49:  50:  51:  52:  53:  54:  55:  56:  57:  58:  59:  60:  61:  62:  63:  64:  65:  66:  67:  68:  69:  70:  71:  72:  73:  74:  75:  76:  77:  78:  79:  80:  81:  82:  83:  84:  85:  86:  87:  88:  89:  90:  91:  92:  93:  94:  95:  96:  97:  98:  99:  100: 101: 102: 103: 104: 105: 106: 107: 108: 109: | public class Catfish {  /\*\*  \* Location of the catfish - which row.  \*/  private int row;  /\*\*  \* Location of catfish - which column  \*/  private int column;  /\*\*  \* Image of the catfish - is really a filename.  \*/  private String imageFileName;  /\*\*  \* Location of catfish - the row  \*  \* @return - an integer representing the row location of catfish.  \*/  public int getRow() {  return row;  }  /\*\*  \* Location of catfish - the column  \*  \* @return - an integer representing the column location of catfish.  \*/  public int getColumn() {  return column;  }  /\*\*  \* Swim one cell to the right.  \* The swimming area will be limited to the lake boundary.  \* If catfish is at the right boundary, it will not swim further right.  \*  \* @return nothing.  \*/  public void swimRightIfPossible() {  // If the cell to the right is within bounds, move right.  if (column + 1 <= 10) {  column = column + 1;  }  imageFileName = "/Catfish-right.gif";  }  /\*\*  \* Swim one cell to the left.  \* The swimming area will be limited to the lake boundary.  \* If catfish is at the left boundary, it will not swim further left.  \*  \* @return nothing.  \*/  public void swimLeftIfPossible() {  // If the cell to the left is within bounds, move left.  if (column - 1 >= 1) {  column = column - 1;  }  imageFileName = "/Catfish-left.gif";  }  /\*\*  \* Swim one cell down.  \* The swimming area will be limited to the lake boundary.  \* If catfish is at the bottom boundary, it will not swim further down.  \*  \* @return nothing.  \*/  public void swimDownIfPossible() {  // If the cell below is within bounds, move down.  if (row + 1 <= 10) {  row = row + 1;  }  imageFileName = "/Catfish-down.gif";  }  /\*\*  \* Swim one cell up.  \* The swimming area will be limited to the lake boundary.  \* If catfish is at the top boundary, it will not swim further up.  \*  \* @return nothing.  \*/  public void swimUpIfPossible() {  // If the cell above is within bounds, move up.  if (row - 1 >= 1) {  row = row - 1;  }  imageFileName = "/Catfish-up.gif";  }  /\*\*  \* get filename of catfish image  \*  \* @return filename of Catfish image  \*/  public String getImage() {  return imageFileName;  }  } |
| Listing 1 *Catfish.java* | |

First of all, let's identify the various components covered in the Java Interlude titled "Identifiers, Statement Order, Format, and Comments."

Listing 1 includes several comments like the following:

/\*\*

\* Swim one cell down.

\* The swimming area will be limited to the lake boundary.

\* If catfish is at the bottom boundary, it will not swim further down.

\*

\* @return nothing.

\*/

Some of the keywords in the listing are:

* public
* class
* return
* if
* else

Some identifiers in the listing are as follows:

* row
* column
* imageFileName
* String
* getRow
* swimRightIfPossible

**Class Definition**

First of all, recall from Section 1.3 of the textbook that to create the Catfish object, which is an instance of the Catfish class, a description for the Catfish class is needed, which can be used as a blueprint for creating Catfish objects. Listing 1 provides the description for the Catfishclass. Hence, it is called the class definition. The description is provided by the source code enclosed within the opening brace at the end of line 1 and the closing brace on line 108. Line 1 starts the description for the Catfish class by conveying some information about the class itself, as indicated by the code public class Catfish.

We will learn more about the meaning of all the identifiers and keywords on the line 1 as we move ahead in the course. For now, please make a note that for many of the classes used by the simulation program, the code on lines 1 and 108 forms the start and end of the class definition, except that the identifier Catfish on line 1 would change to be the name of the respective class.

Recall from [2.1.1 Programming with Objects](javascript:ContentByName('pg-objects-java')) that a class describes the attributes and behavior for its objects. Let us examine the listing further to learn more about the attributes and behavior for the Catfishobjects.

**Attributes of Catfish Class**

The statements on lines 6, 11, and 16 describe the attributes for objects of the Catfishclass. The statements on lines 6 and 11 describe the attributes that correspond to the row and column, respectively, in the grid for the location of the fish. The statement on line 16 describes the attribute that corresponds to the name of the image file used to represent the fish at any given time.

Note that each of these three statements consists of three words the first of which is common: private. This is a keyword in Java indicating that these attributes can only be accessed by other Catfishobjects. We will learn more about this keyword in [2.2 Fundamentals of Object-Oriented Programming](javascript:ContentByName('pg-oop-fund')).

The words int and String are keywords in Java that indicate the type of values the attributes can take. Thus, the attributes row and column can take integer values, whereas the name of the image file must be a String object. We will learn more about these and other such keywords further in [2.3 Fundamentals of Java](javascript:ContentByName('pg-java-fund')).

The last word on each of these statements is the actual name of the attribute: row, column, and imageFileName.

Thus, the statements on lines 6, 11, and 16 are declaration statements for the attributes of the Catfish class. As covered in the Java Interlude titled "Variables and Assignment" from the textbook, the declaration for each attribute includes a type and an identifier. In addition, each of these declarations includes a keyword that specifies the access to the attribute.

**Behavior of Catfish Class**

The remaining source code in the listing, other than the comments, is part of the description for the behavior of Catfishobjects. It describes how the Catfishobject would respond to the following messages:

* getRow
* getColumn
* getImage
* swimRightIfPossible
* swimLeftIfPossible
* swimUpIfPossible
* swimDownIfPossible

Let us look at the description for the swimRightIfPossible message. Lines 43–50 describe the response of the Catfishobjects to this message. The opening brace on line 43 is paired with a closing brace on line 50. The lines between the two braces collectively constitute the *method definition* for the swimRightIfPossible message. We will learn more about method definitions in [2.2 Fundamentals of Object-Oriented Programming](javascript:ContentByName('pg-oop-fund')).

Note that line 43 has three words, just like the statements for the attributes we saw above, followed by a set of empty parentheses "()" and the opening brace "{". The empty parentheses indicate that this message does not require any resources to complete execution. As you work with other objects in this course, you will learn that some messages do require resources to be sent by the object that initiates the message.

The first word on line 43 is a Java keyword: public. This keyword indicates that any object can send this message to *Catfish* objects in a program. We will learn more about this keyword in [2.2 Fundamentals of Object-Oriented Programming](javascript:ContentByName('pg-oop-fund')).

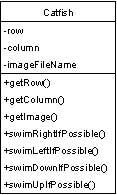
The second word on line 43 is another Java keyword void. This keyword indicates that the swimRightIfPossible message returns no value after it completes execution. We will learn more about this and other such keywords further in [2.3 Fundamentals of Java](javascript:ContentByName('pg-java-fund')).

The last word on line 43 is an identifier: swimRightIfPossible. This identifier is the name of the message.

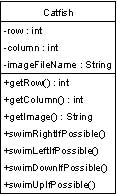
The following observations can help you examine the method definitions for the other messages in the listing:

* The names of messages fall into two categories. One set follows the get\* pattern, while the other follows the swim\*IfPossible pattern. These names should be self-explanatory as to the behavior of the respective method definitions. The first set of messages allows any other objects that communicate with the Catfishobjects to get information about their attributes. For instance, the getRow message allows a communicating object to find out the row in the grid corresponding to the location of the Catfishobject. The second set of messages attempts to modify a Catfishobject to swim in a particular direction. For instance, the swimRightIfPossible message will describe the behavior of Catfishobjects that attempt to swim to the right in the grid.
* The names of the messages follow a certain convention, where the name actually uses words that convey the intent of the behavior. The first letter of each word but the first is capitalized. This is a recommended Java programming convention. We will learn more about other such programming conventions later in the course. For now, please make a note: following such conventions in the source code makes it readable and easy to understand. Even though we did not write this source code for Catfishclass ourselves, we can understand parts of it simply by reading it.
* The messages that follow the get\* naming pattern return something after completing execution, while those that follow the swim\*IfPossible naming pattern do not return any value.

Following the convention we used in [2.1.1 Programming with Objects](javascript:ContentByName('pg-objects-java')), we can represent the Catfishclass from Listing 1 as follows:



Actually, we can augment this diagram as follows. The source code has given us more information about the type of values the attributes can take and the type of values the messages return which can be added to the diagram as follows:



**Student Activity**

With the detailed knowledge of the Catfish class from this section, and knowing how to develop a Java class from [2.1.2 Java Program Development](javascript:ContentByName('pg-servlet-develop')), here is something you can try:

* Download [Alife-basic.zip](http://www.icarnegie.com/content/SSD/SSD1/2.1/normal/pg-intro-java-oop/pg-prog-java/pg-class-elements_v2/Alife-basic.zip) and load the various .gif files and the HTML form in CreateFishes.html into the iCarnegie Servlet Workbench. Next, load the various .class files into the workbench and run the simulation program following instructions from [2.1.2 Java Program Development](javascript:ContentByName('pg-servlet-develop')).
* Copy the source code from Listing 1 into *Catfish.java* file. Remember that the line numbers in the listing are just for your convenience, and should not be copied into the source file.
* Edit *Catfish.java* to use a new set of images for the fish. Once you have located the image files of your choice for the fish, edit the lines 49, 65, 81 and 97 to use your image files instead of the following: Catfish-right.gif, Catfish-left.gif, Catfish-down.gif, and Catfish-up.gif.
* Compile *Catfish.java* following instructions from [2.1.2 Java Program Development](javascript:ContentByName('pg-servlet-develop')). You now have a new version of the *Catfish* class in the file *Catfish.class*.
* Load the new version of *Catfish.class* created above into the workbench. This will overwrite the *Catfish.class* file that you previously loaded from Alife-basic.zip.
* Run the new version of the simulation program following instructions from [2.1.2 Java Program Development](javascript:ContentByName('pg-servlet-develop')).

**If-Statement**

When an object gets a message, all statements in the message are executed sequentially in the order in which they appear. Sometimes, we do not want *all* statements to be executed. Instead, we want some statements to be executed only when certain conditions hold true. An if-statement allows us to accomplish such conditional execution.

|  |  |  |
| --- | --- | --- |
| **Execution sequence of an if-statement** | **Example: From swimRightIfPossible**  **(see Listing 1 above, starting on line 43)** | **Example expressed in Java code** |
|  |  | if (column + 1 <= 10) {  column = column + 1;  } |

In the swimRightIfPossible method (see Listing 1, line 43), we say that swimming right is conditional upon catfish staying within bounds. Therefore, we use an if-statement to execute the statement to swim right conditionally. The if-statement checks if the new column value will be less than or equal to 10 (the lake boundary). The check is expressed within parentheses: (column + 1 <= 10) (see line 46). Swimming right is accomplished by incrementing the value of column attribute: column = column + 1; (see line 47). The block of statements that would be conditionally executed are surrounded by curly braces: {}. In our case, the block of statements consists of only one statement (line 47).

See section 2.8 of the Arnow, Dexter, and Weiss textbook—which is required reading—for more examples of if-statements. To further your understanding of if-statements, you might review the examples in that section and imagine how they might be changed—to give different results. To get started, how would you modify the first example on page 58 to display a message "TV strongly prohibited" if age is less than 3?

In this section, we have looked at the source code of a simple Java class in some detail. In the process, we have looked at how a class definition in Java describes the attributes and behavior of an object. We have also seen how the class definition can be modified. Finally, we discussed how we can use if-statements to execute some statements only if certain conditions are true.

**2.1.4 Elements of a Java Servlet**

This section will further apply the knowledge you have gained from the readings in Chapter 1 of the textbook to examine the source code for the Welcomeservlet in some detail. Some of the concepts covered in this section will look familiar from [2.1.3 First Look at Java](javascript:ContentByName('pg-class-elements')). We will also examine a couple of variations of the Welcome servlet.

The first variation has the Welcome servlet prepare a Web page to be sent to the browser instead of plain text. The next variation presents a more personalized welcome message for display, involving the user's name typed into the form invoking the servlet. Note that the servlet code we examine in this section is still simple at this time and hence will not handle any possible errors. For instance, when we look at its variation for a personalized welcome message, the message will not look different in the event of errors, such as the user submitting the form without typing a name.

**The Welcome Servlet**

The source code for the Welcome servlet from the servlet source file in [2.1.2 Java Program Development](javascript:ContentByName('pg-servlet-develop')) is shown in Listing 1.

|  |  |
| --- | --- |
| 1:  2:  3:  4:  5:  6:  7:  8:  9:  10: 11: 12: 13: 14: 15: 16: 17: 18: 19: 20: 21: 22: 23: 24: 25: 26: 27: 28: 29: 30: 31: | /\*\*  \* Author: ssd1-dev-srt  \* Date: May'03  \* Description: Servlet that displays a welcome message.  \*/  import java.io.\*;  import javax.servlet.\*;  import javax.servlet.http.\*;  public class Welcome extends HttpServlet {  public void doPost(HttpServletRequest request,  HttpServletResponse response)  throws ServletException, IOException {  /\*\*  \* Indicate the content type (for example, text/html),  \* being returned by the response  \*/  response.setContentType("text/plain");  /\*\*  \* Retrieve an output stream to use to send data to the client  \*/  PrintWriter out = response.getWriter();  /\*\*  \* Write the response  \*/  out.println("Welcome to iCarnegie!");  }  } |
| Listing 1 *Welcome.java* | |

First, let's identify the various components we read about in the Java Interlude titled "Identifiers, Statement Order, Format, and Comments."

Listing 1 includes comments on lines 1–5, 15–18, 21–23, and 26–28.

Further, the statements in the Welcomeservlet are as follows:

import java.io.\*;

import java.servlet.\*;

import java.servlet.http.\*;

response.setContentType("text/plain");

PrintWriter out = response.getWriter();

out.println("Welcome to iCarnegie!");

Some of the keywords in the listing are:

public

class

extends

void

throws

import

The listing also includes several identifiers as follows:

Welcome

HttpServlet

doPost

HttpServletRequest

HttpServletResponse

PrintWriter

ServletException

IOException

request

response

out

setContentType

getWriter

println

Let us examine each of the statements we have identified so far.

Lines 6–8 are statements known as *import directives*, as you have read in the Java Interlude titled "Packages and the import Statement" of the textbook. These lines make pre-defined code from the packages java.io and javax.servlet available to the servlet. In particular, line 6 imports all the classes contained in the package java.io, which are required by the servlet to do any input or output tasks. Lines 7 and 8 import *all* classes in the packages javax.servletand javax.servlet.httprespectively.

First of all, to create the Welcome servlet, which is an instance of the Welcome class, a description for the Welcome class is needed that can be used as a blueprint for creating Welcome objects. This is done by the source code enclosed within the opening brace at the end of line 10 and the closing brace on line 31. Line 10 starts the description for the Welcome class by conveying some information about the class itself, as indicated by the code public class Welcome extends HttpServlet. We will learn more about the meaning of all the identifiers and keywords on line 10 as we move ahead in the course. For now, please note that, for all the servlets we work with, lines 10 and 31 would form the start and end of the description for the servlet class in our source code, except that the identifier Welcome on line 10 would change to the name of the servlet we are writing.

Next, recall that, if the form that invokes the Welcome servlet has the **method** attribute set to **post**, the servlet receives a doPost message when the form is submitted. So, the description for the Welcome class needs to include information on how a doPostmessage will be processed. This is indicated by the source code enclosed within the opening brace at the end of line 11 and the closing brace on line 30. Lines 11–13 start the description for the doPostmessage by expressing information about the kinds of objects that can send this message to the servlet, the resources needed by the message to execute, and the types of errors that may occur during its execution.

In particular, the resources needed by the doPostmessage are specified by the code: HttpServletRequest request, HttpServletResponse response. This code indicates that request and response are objects of the HttpServletRequest and HttpServletResponse classes, respectively. These are pre-defined classes we imported in lines 7 and 8. When the doPostmessage is received by the servlet, the request object enables the servlet to access all the information related to the browser request that invoked the servlet, including user input from the HTML form. The response object enables the servlet to send its response back to the browser. We will learn more about the meaning of all other identifiers and keywords on lines 11–13 as we move ahead in the course. For now, please note that, for all the servlets we work with, lines 11–13 and line 30 would form the start and end of the description for the doPost message in our source code.

The first step in processing the doPostmessage is to inform the browser that it should expect plain text from the server, as the comment on lines 15–18 indicates. For this, the servlet sends the setContentTypemessage to the responseobject as seen on line 19. The setContentTypemessage needs one resource to execute, and this resource is specified by the code "text/plain" indicating that the servlet will generate plain text. On the other hand, a servlet that generates an HTML Web page should specify "text/html" when sending the setContentTypemessage, while a servlet that generates a GIF file would specify "text/gif" instead.

The next step in processing the doPostmessage indicated by the statement on line 24 is to send the getWritermessage to the *response* object. Note that getWritermessage needs no resources for execution as indicated by the empty parentheses following the identifier getWriteron line 24. Also note that the getWritermessage returns an object of the PrintWriter class identified by out after it completes execution. The PrintWriter class is another pre-defined class we imported; it processes messages to output text to the browser.

The final step in processing the doPostmessage, as indicated by the statement on line 29, is to send the println message to the out object obtained in the previous step. The println message requires one resource to execute, which is the actual text that must be displayed in the browser window. This is indicated by the code, "Welcome to iCarnegie!"

With the knowledge of the Welcomeservlet from this section, and knowing how to develop a Java servlet from [2.1.2 Java Program Development](javascript:ContentByName('pg-servlet-develop')), you are ready to start writing Java servlets. Here is something you can try:

* Copy the source code from Listing 1 into *Welcome.java* file. Remember that the line numbers in the listing are just for your convenience and should not be copied into the source file.
* Edit *Welcome.java* to make some change in the text message to be displayed in the browser. Essentially, you need to change the parameter for the println method on line 31. Make sure the message is still enclosed in quotes (" ").
* Compile *Welcome.java* following instructions in [2.1.2 Java Program Development](javascript:ContentByName('pg-servlet-develop')). You now have a new version of the Welcome servlet in the file *Welcome.class*, which displays a different welcome message.
* Run the new version of the Welcome servlet following instructions in [2.1.2 Java Program Development](javascript:ContentByName('pg-servlet-develop')). Note that you will need to load the new version of *Welcome.class* into the iCarnegie Servlet Workbench and that you can use the same form that was used in [2.1.2 Java Program Development](javascript:ContentByName('pg-servlet-develop')).

**More Servlets**

Let's now look at a variation of the Welcome servlet. Instead of plain text, this variation will send an HTML Web page to the browser.

The first thing to do is to ensure the browser knows about the change. As seen earlier in the description of line 19 of listing 1, we need to specify the parameter "text/html"instead of "text/plain"when sending the setContentTypemessage to the responseobject. This line would appear as:

response.setContentType("text/html");

To send an HTML Web page from the servlet, the servlet essentially needs to output all the HTML code for the Web page to the browser. Writing the HTML code for the Web page first is helpful. You now know from Unit 1 that for a Web page that displays our welcome message from listing 1, the HTML code would be as follows:

<!DOCTYPE HTML PUBLIC '-//W3C//DTD HTML 4.01 Transitional//EN'>

<HTML>

<HEAD>

<TITLE>

Welcome

</TITLE>

</HEAD>

<BODY>

Welcome to iCarnegie!

</BODY>

</HTML>

To send a Web page, the servlet essentially needs to send the printlnmessage to the out object for every line of HTML code to be sent to the browser. This amounts to using the statement from line 31 of listing 1 on each line of HTML code for the Web page. When this is done, the source code for the servlet from listing 1 now looks as follows:

|  |  |
| --- | --- |
| 1:  2:  3:  4:  5:  6:  7:  8:  9:  10: 11: 12: 13: 14: 15: 16: 17: 18: 19: 20: 21: 22: 23: 24: 25: 26: 27: 28: 29: 30: 31: 32: 33: 34: 35: 36: 37: 38: 39: 40: 41: | /\*\*  \* Author: ssd1-dev-srt  \* Date: May'03  \* Description: Servlet that displays a welcome message.  \*/  import java.io.\*;  import javax.servlet.\*;  import javax.servlet.http.\*;  public class HtmlWelcome extends HttpServlet {  public void doPost(HttpServletRequest request,  HttpServletResponse response)  throws ServletException, IOException {  /\*\*  \* Indicate the content type (for example, text/html),  \* being returned by the response  \*/  response.setContentType("text/html");  /\*\*  \* Retrieve an output stream to use to send data to the client  \*/  PrintWriter out = response.getWriter();  /\*\*  \* Write the response  \*/  out.println(  "<!DOCTYPE HTML PUBLIC '-//W3C//DTD HTML 4.01 Transitional//EN'>");  out.println("<HTML>");  out.println("<HEAD>");  out.println("<TITLE>");  out.println("Welcome");  out.println("</TITLE>");  out.println("<BODY>");  out.println("Welcome to iCarnegie!");  out.println("</BODY>");  out.println("</HTML>");  }  } |
| Listing 2 *HtmlWelcome.java* | |

Note that line 10 now indicates that the name of the servlet is HtmlWelcome. Essentially, we have created a new servlet.

If you have already worked through the servlet development instructions so far, here is another task for you to try:

* Copy the source code from Listing 2 into *HtmlWelcome.java* file. Again, note that the line numbers in the listing are just for your convenience and should not be copied into the source file.
* Compile *HtmlWelcome.java* following instructions in [2.1.2 Java Program Development](javascript:ContentByName('pg-servlet-develop')). You now have a new servlet HtmlWelcomein the file *HtmlWelcome.class,* which sends the same welcome message as the servlet Welcomebut sends it as part of a Web page instead of as plain text.
* We now need a new form to invoke the HtmlWelcome servlet, which can be a copy of the form we used to invoke the Welcome servlet. So, make a copy of *WelcomeForm.html* file and name it *HtmlWelcomeForm.html*. Edit the file *HtmlWelcomeForm.html* so that the **action** attribute of the **FORM** element in that file invokes the HtmlWelcome servlet instead of the Welcome servlet.
* Run the HtmlWelcome servlet following instructions in [2.1.2 Java Program Development](javascript:ContentByName('pg-servlet-develop')). Note that you will need to load the files *HtmlWelcome.class* and *HtmlWelcomeForm.html* into the iCarnegie Servlet Workbench.

The message displayed in the browser will not look different when the forms in *WelcomeForm.html* and *HtmlWelcomeForm.html* are submitted. However, if you pull-down the View menu of the browser and select the Source option in both cases, the first case will display just the welcome message Welcome to iCarnegie! and the second case will actually display the HTML code as we wrote before editing the Welcome servlet:

<!DOCTYPE HTML PUBLIC '-//W3C//DTD HTML 4.01 Transitional//EN'>

<HTML>

<HEAD>

<TITLE>

Welcome

</TITLE>

</HEAD>

<BODY>

Welcome to iCarnegie!

</BODY>

</HTML>

Thus, when writing servlets that send a Web page to the browser, it is a good idea to implement the HTML code for the Web page before writing the out.println statements in the servlet source file. For example, consider the case where, the servlet sends an image instead of displaying a welcome message. The HTML code for the Web page would now look something like:

<!DOCTYPE HTML PUBLIC '-//W3C//DTD HTML 4.01 Transitional//EN'>

<HTML>

<HEAD>

<TITLE>

Welcome

</TITLE>

</HEAD>

<BODY>

<IMG src = '/sf.jpg' alt='San Francisco'>

</BODY>

</HTML>

So, line 37 in Listing 2 would need to be changed. Following the rule that the text to be displayed by the println method is to be enclosed in quotes (" "), the new line 37 should be:

out.println("<IMG src='/sf.jpg' alt='San Francisco'>");

If you have worked through the HtmlWelcomeexample, here is something else you can try:

* Edit *HtmlWelcome.java* to display an image instead of a text message. So, change the line 37 as above. Note that we use the image file *sf.jpg* here. You may use any image file available to you. Make sure you change the specification of the **alt** attribute accordingly. Again, it is a good idea to write the HTML code to be sent by the servlet before we changing the servlet source file.
* Compile *HtmlWelcome.java* following instructions in [2.1.2 Java Program Development](javascript:ContentByName('pg-servlet-develop')). You now have a new version of the servlet HtmlWelcomein the file *HtmlWelcome.class* that sends an image as part of a Web page instead of a text message.
* Run the new version of the HtmlWelcome servlet following instructions in [2.1.2 Java Program Development](javascript:ContentByName('pg-servlet-develop')). Note that you will need to load the new version of *HtmlWelcome.class* into the iCarnegie Servlet Workbench. You will also need to load the image file that you are using into the HTML directory of the workbench, if not already done. You can use the same form that was used for the earlier version of the HtmlWelcomeservlet.

**Getting User Input**

While the form in *WelcomeForm.html* allowed the user to enter a name, the message sent by the Welcomeservlet had nothing to do with the name entered by the user. This is because the source code for the Welcomeservlet we examined did not attempt to find out what the user entered and to include that information in the Web page sent to the browser.

When a servlet is invoked by a form, there is a way for the servlet to obtain the user input specified in the form. To do this, the servlet needs to send the getParametermessage to request, which is an HttpServletRequest object. Thus, we would need to add a new line of code to our listing 1 as follows:

String userName = request.getParameter("Name");

When the request object processes the getParameter message, the result is an instance of the String class identified by the name userName. The getParameter message takes the parameter "Name", which identifies the name of the input text control in the form in *WelcomeForm.html*.

We also need to modify line 29 of Listing 1 to display the name in the output HTML. The new line 29 would be:

out.println("Welcome to iCarnegie, " + userName + "!");

Note that the parameter to the println message sent to the out object has now changed. Instead of one pair of quotes enclosing the welcome message, it now has three parts separated by the "+" character: "Welcome to iCarnegie, ", userName, and "!". Here, we create a new string by applying *string concatenation* covered in Section 2.7 of the textbook. In this particular case, we are interested in making a string that looks like:

Welcome to iCarnegie, John Doe!

where the name "John Doe" is the value obtained using the getParameter statement earlier. Thus, the statement above builds a string by concatenating or joining smaller strings using the "+" operator and substituting the user's name within the string.

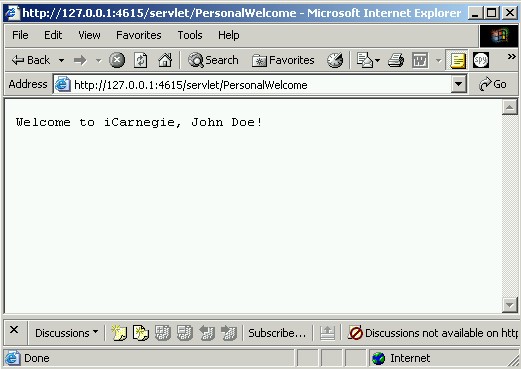
Listing 3 shows the source code for a new servlet, PersonalWelcome. Most of it looks very similar to the listing 1 for the Welcomeservlet, except that PersonalWelcomesends a personalized message to the browser with the user's name obtained from the form that invokes it as in line 34.

|  |  |
| --- | --- |
| 1:  2:  3:  4:  5:  6:  7:  8:  9:  10: 11: 12: 13: 14: 15: 16: 17: 18: 19: 20: 21: 22: 23: 24: 25: 26: 27: 28: 29: 30: 31: 32: 33: 34: 35: 36: 37: | /\*\*  \* Author: ssd1-dev-srt  \* Date: May'03  \* Description: Servlet that displays a personalized welcome message.  \*/  import java.io.\*;  import javax.servlet.\*;  import javax.servlet.http.\*;  public class PersonalWelcome extends HttpServlet {  public void doPost(HttpServletRequest request,  HttpServletResponse response)  throws ServletException, IOException {  /\*\*  \* Indicate the content type (for example, text/html),  \* being returned by the response  \*/  response.setContentType("text/plain");  /\*\*  \* Retrieve an output stream to use to send data to the client  \*/  PrintWriter out = response.getWriter();  /\*\*  \* Get the user input from the form  \*/  String userName = request.getParameter("Name");  /\*\*  \* Write the response  \*/  out.println("Welcome to iCarnegie, " + userName + "!");  }  } |
| Listing 3 *PersonalWelcome.java* | |

Here is one more task you can try:

* Copy the source code from Listing 3 into *PersonalWelcome.java* file. Again, note that the line numbers in the listing are just for your convenience and should not be copied into the source file.
* Compile *PersonalWelcome.java* following instructions in [2.1.2 Java Program Development](javascript:ContentByName('pg-servlet-develop')). You now have a new servlet PersonalWelcomein the file *PersonalWelcome.class,* which sends the same welcome message as the servlet Welcomebut also includes the user's name.
* We now need a new form to invoke the PersonalWelcome servlet, which can be a copy of the form we used to invoke the Welcome servlet. So, make a copy of *WelcomeForm.html* file and name it *PersonalWelcomeForm.html*. Edit the file *PersonalWelcomeForm.html* so that the **action** attribute of the **FORM** element in that file invokes the PersonalWelcome servlet instead of the Welcome servlet.
* Run the PersonalWelcome servlet following instructions in [2.1.2 Java Program Development](javascript:ContentByName('pg-servlet-develop')). Note that you will need to load the files *PersonalWelcome.class* and *PersonalWelcomeForm.html* into the iCarnegie Servlet Workbench.

The following screenshot shows what your browser display may look like:

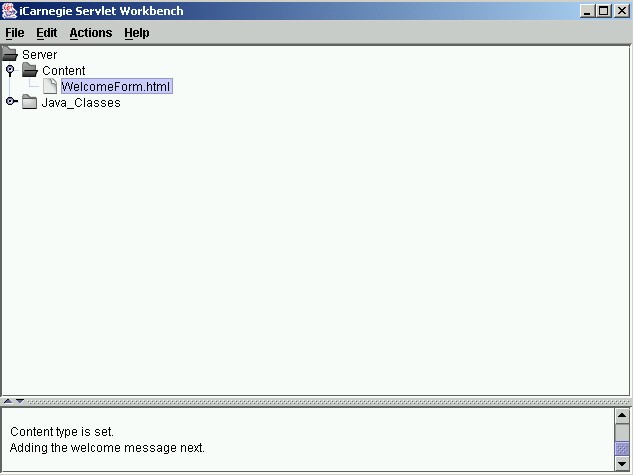


**Debugging a Servlet**

Sometimes, you may need to monitor the execution of a servlet. This would be useful when you need to find an error in the servlet code. This error would be an execution-time error or a run-time error as covered in Section 1.8 of the textbook.

|  |  |
| --- | --- |
| 1:  2:  3:  4:  5:  6:  7:  8:  9:  10: 11: 12: 13: 14: 15: 16: 17: 18: 19: 20: 21: 22: 23: 24: 25: 26: 27: 28: 29: 30: 31: 32: 33: 34: 35: 36: 37: 38: 39: 40: 41: 42: | /\*\*  \* Author: ssd1-dev-srt  \* Date: May'03  \* Description: Servlet that displays a welcome message.  \*/  import java.io.\*;  import javax.servlet.\*;  import javax.servlet.http.\*;  public class Welcome extends HttpServlet {  public void doPost(  HttpServletRequest request,  HttpServletResponse response)  throws ServletException, IOException {  /\*\*  \* Indicate the content type (for example, text/html),  \* being returned by the response  \*/  response.setContentType("text/html");  // Debug line  System.out.println(" Content type is set.");  /\*\*  \* Retrieve an output stream to use to send data to the client  \*/  PrintWriter out = response.getWriter();  // Debug line  System.out.println(" Adding the welcome message next.");  /\*\*  \* Write the response  \*/  out.println("Welcome to iCarnegie!");  }  } |
| Listing 4 *Welcome.java* | |

Listing 4 shows the source code for the Welcome servlet that has additional System.out.println statements on lines 24 and 34. These two statements display information about how far the servlet execution has progressed in the console window of the workbench as seen in the following screenshot. Note that these two statements use System.out.println instead of out.println. As covered in Section 1.5 of the textbook, both statements send a println message to the System.out object, with a String that specifies the information to be displayed.



In this section, we examined the source code of a simple servlet in detail. This essentially involved understanding how the servlet processes a doPost message. We also saw some variations of this servlet, such as sending a Web page with an image, and obtaining user input from the form that invokes the servlet. In the process, we saw how servlets generate HTML pages dynamically (that is, generating HTML based on input from the user). Examining the different variations of the servlet has actually shown how we can change the way the doPost message is processed by the servlet. In the next section, we will work through an example that uses all these variations.

**2.1.5 Planning Servlet Development**

We have looked at the Java code in a very simple servlet in some detail. We have also learned how to edit, compile, and run a Java servlet. This information, along with our knowledge of HTML forms from [1.3 Introduction to HTML Forms and Servlets](javascript:ContentByName('pg-forms-servlets')), is sufficient to get us started on writing our own servlet that processes user input from an HTML form and generates a Web page in response.

When working through the assessments in [1.3 Introduction to HTML Forms and Servlets](javascript:ContentByName('pg-forms-servlets')), we learned how to create an HTML form from an HTML form specification. In this section, we will learn to work from a similar specification for writing a servlet. Typically, this specification will provide the following information about the servlet:

* The user input(s) sent from the HTML form
* The layout and contents of the Web page to be generated by the servlet in response, after processing the user input(s)

Let us consider the task of developing a Java servlet to handle one of the forms from the web site of a travel agency. Users seeking information on some travel destination would be presented this form when they visit the agency's web site. The form allows users to input their name, phone number and a choice of travel destination. The servlet invoked by the form responds with a Web page that displays a confirmation of the user's request for information and a picture of the destination selected. To get us started, we are given a form in an HTML file and a *.java* template file to be used for writing the servlet.

**Thinking Ahead**

As you start thinking about writing the servlet, you first need to make sure that the following is applicable to your PC:

* The J2SE SDK is correctly installed on your PC and you are able to compile a Java servlet. Note that this requires that the file *javax.servlet.jar* from the iCarnegie Servlet Workbench installation be added to your **classpath**. Please refer to [Appendix B. The iCarnegie Servlet Workbench](javascript:ContentByName('pg-ic-servlet-workbench-appendix');) for information on how to do this.
* The iCarnegie Servlet Workbench is correctly installed on your PC, and you have completed the assessments in [1.3 Introduction to HTML Forms and Servlets](javascript:ContentByName('pg-forms-servlets')). Hence, you should be familiar with using the iCarnegie Servlet Workbench to load HTML forms and servlet class files. You should also be familiar with opening and submitting the HTML forms using the workbench, to invoke the servlets on the Web server included with the workbench installation.

As you embark on programming, please note that giving in to the temptation to start writing code after a quick look at the specification, or "jumping into coding" as it is commonly known, is not a good habit for a programmer. More often than not, it can lead to more programming time and effort, and usually not to the best solution. This is due to lack of planning and structure.

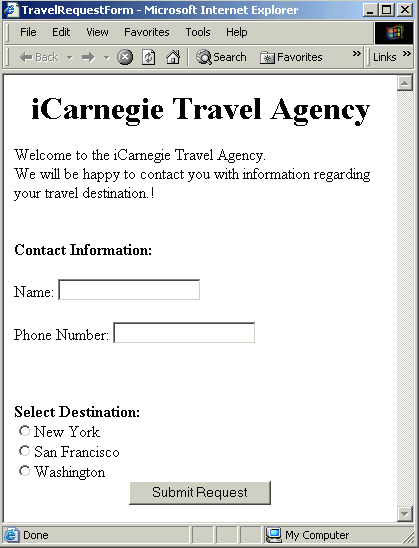
**Defining the Problem**

As part of the specification for the problem, we are given an HTML form in *TravelRequestForm.html* that refers to three image files. The form contains the following elements:

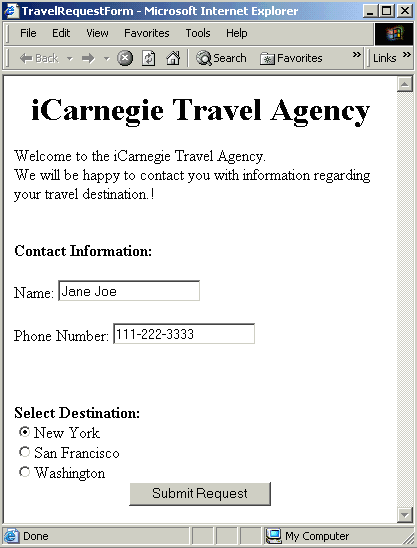
* A single-line text input control named *Name*
* A single-line text input control named *PhoneNumber*
* A radio button input control named *Destination* that has three options:
  + New York
  + San Francisco
  + Washington
* A submit button labeled *Submit Request*

Further, the **FORM** element in *TravelRequest.html* uses the **action** '/servlet/TravelRequest' and the **post** method.

A sample screenshot of such a form follows:



A sample screenshot of the request submitted by a typical user is as follows:

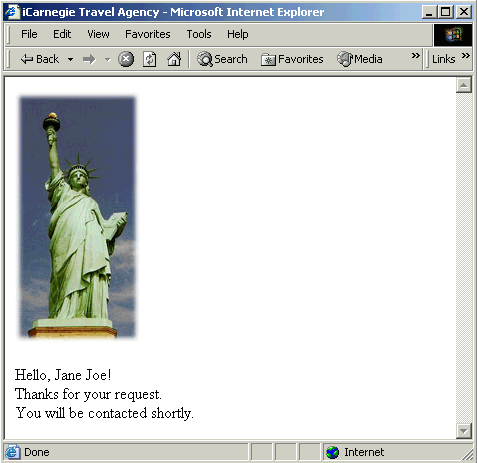


Our goal is to develop a servlet called TravelRequest that processes user input from the form in *TravelRequestForm.html*. The servlet responds with an HTML Web page that displays the following information:

* A confirmation message that includes the user's name obtained from the form, and informs the user that he or she will be contacted soon
* An image of some landmark from the travel destination selected by the user on the form

At this time, please download [TravelRequest.zip](http://www.icarnegie.com/content/SSD/SSD1/2.1/normal/pg-intro-java-oop/pg-prog-java/pg-servlet-plan_v2/TravelRequest.zip) to the directory where you plan to work. Then, extract the contents of the zip file. The directory named *work* contains the skeleton Java source file (*.java*) for the TravelRequest servlet. The directory named *content* contains the HTML file and image files necessary to run the servlet.

A sample screenshot of the Web page to be generated by the servlet is as follows:



We are also provided a skeleton *.java* file that can be used as a template for coding the servlet. This file has comments that guide us through the various tasks that the servlet needs to do. It also has some of the servlet code already written for us.

Listing 1 shows the template file that we are provided.

|  |  |  |
| --- | --- | --- |
| 1:  2:  3:  4:  5:  6:  7:  8:  9: 10: 11: 12: 13: 14: 15: 16: 17: 18: 19: 20: 21: 22: 23: 24: 25: 26: 27: 28: 29: 30: 31: 32: 33: 34: 35: 36: 37: 38: 39: 40: 41: 42: 43: 44: 45: 46: 47: 48: 49: 50: 51: 52: 53: 54: 55: 56: 57: 58: 59: 60: 61: 62: 63: 64: | | /\*\*  \* Author:  \* Date:  \* Description:  \*  \*/  import java.io.\*;  import javax.servlet.\*;  import javax.servlet.http.\*;  public class TravelRequest extends HttpServlet {  public void doPost(  HttpServletRequest request,  HttpServletResponse response)  throws ServletException, IOException {  /\*\*  \* Indicate the content type (for example, text/html),  \* being returned by the response  \*/  response.setContentType("text/html");  /\*\*  \* Retrieve an output stream to use to send  \* data to the client  \*/  PrintWriter out = response.getWriter();  /\*\*  \* Get the user input from the form : name and destination  \*/  /\* Add your code here \*/  /\*\*  \* Start by building the web page header  \*/  /\* Add your code here \*/  /\*\*  \* Add the image  \*/  /\* Add your code here \*/  /\*\*  \* Add the confirmation message, with the name  \*/  /\* Add your code here \*/  /\*\*  \* End by building the web page footer  \*/  /\* Add your code here \*/  } // end-doPost  } |
| **Listing 1** *TravelRequest Template File* | | |

**Planning the Solution**

From the comments in the *.java* template file, we conclude that the servlet needs to do the following tasks:

1. Indicate the content type being returned by the response
2. Retrieve an output stream to send data to the client
3. Get user input from the form: name and destination
4. Start by building the Web page header
5. Add the image
6. Add the confirmation message, with the name
7. End by building the Web page footer

From examining the *.java* template file above, we see that a few lines of code or statements are already provided for steps 1 and 2. So, we only need to write code for steps 3 through 7. From the servlet examples seen so far, we know that this part of the servlet will construct the Web page to be returned to the browser. Clearly, the two major components of this Web page are a confirmation message including the name entered by the user, which depends on one of the text input controls of the form in *TravelRequestForm.html*, and an image, which depends on the radio button input control of the form in *TravelRequestForm.html*. Since the Web page generated by the servlet does not display the phone number entered by the user, the servlet need not worry about obtaining that information from the form. Hence, the comment in the template file mentions only name and destination. The servlet needs to get only two of the three user inputs from the form: the name and the destination.

Let us first think about how to obtain these inputs in our servlet.

* We know from the PersonalWelcomeservlet example of [2.1.4 Elements of a Java Servlet](javascript:ContentByName('pg-servlet-elements')) how to obtain user input from an HTML form by sending the getParameter message to the HttpServletRequest object. First, the radio button input control on the form has options that are names of cities. The servlet needs to know which city was selected by the user and then pick the respective image file to include in the Web page it generates. We know from [1.3 Introduction to HTML Forms and Servlets](javascript:ContentByName('pg-forms-servlets')) how a radio button input control works in a form. The HTML code for the form in *TravelRequestForm.html* shows that the value of the radio input control called *Destination* returns the user's choice of destination city. It also shows that selecting *New York*, *San Francisco*, or *Washington* on the form actually returns the values *ny.jpg*, *sf.jpg,* or *wdc.jpg*, respectively. Our knowledge of [2.1.4 Elements of a Java Servlet](javascript:ContentByName('pg-servlet-elements')) tells us that sending the getParameter message to the HttpServletRequest object to obtain the value of the radio button input control *Destination* from the form actually provides the name of the image file for the destination city selected.
* Next, the HTML code for the form in *TravelRequestForm.html* shows that the text input control called *Name* returns the name of the user. Again, to get the name of the user from the form, we need to send the getParameter message to the HttpServletRequest object to get the value of the text input control *Name*.

Let us now think about how the servlet would generate the Web page based on the user inputs obtained from the form.

* From [2.1.4 Elements of a Java Servlet](javascript:ContentByName('pg-servlet-elements')), to build a Web page, the servlet must output all the HTML code for the Web page. To display the image for the destination city, the HTML code needs an IMG tag that uses the specification of the image file obtained by processing the user input to the form. Further, to display the confirmation message with the user's name, the HTML code must include a line of text that is the concatenation of the message text with the user's' name.
* From the HtmlWelcomeexample of [2.1.4 Elements of a Java Servlet](javascript:ContentByName('pg-servlet-elements')), we know that, to generate a Web page, the servlet needs an out.printlnstatement per line of HTML code of the Web page. To include the confirmation message with the user's name, we must construct a message that includes the name obtained from the form. Again, we know how to do this using string concatenation.

Before we start coding, let us see if we can apply the *divide-and-conquer* technique to our programming task here. We need to see if we can divide our task into smaller increments, each of which can be individually tested for correctness before moving to the next. On the one hand, there seem to be three major areas to be handled by the servlet: obtaining user input from the form, processing this input further if necessary to prepare for the output and preparing the output in the form of a Web page. We could code the servlet using these three steps, and then try to test it at the end of each step to see if what we wrote actually works correctly. However, there isn't a good way to test the first two, since the servlet would not send any response to the browser at the end of each step. Since viewing a response Web page is the only way we can confirm that the servlet is working correctly, constructing a Web page that partially mimics the specification of the response Web page would be a good first step. We could then follow up with the actual user inputs in two later steps.

Let us then plan on writing the code in the following steps:

* Write the code to build the Web page with a header, a footer, and a confirmation message built with some dummy name using string concatenation. Test by opening the form in *TravelRequestForm.html* using the workbench, submit the form, and verify that the Web page displays correctly.
* Write the code to obtain the user's choice of destination city and to construct a Web page with the image for the destination city. Again, test by opening the form in *TravelRequestForm.html* using the workbench, submit the form, and verify that the Web page displays the image for the destination city selected.
* Write the code to obtain the user's name and substitute that for the dummy name in the confirmation message. As before, test by opening the form in *TravelRequestForm.html* using the workbench, submit the form, and verify that the Web page displays the name as entered in the form.

As we code the increments laid out above, make a note that each step involves the following:

* Edit the servlet source file *TravelRequest.java* to include the code for that step.
* Compile the servlet source file using the *javac* command, as in [2.1.2 Java Program Development](javascript:ContentByName('pg-servlet-develop')) and [2.1.4 Elements of a Java Servlet](javascript:ContentByName('pg-servlet-elements')).
* Run the servlet using the form in *TravelRequestForm.html* and the iCarnegie Servlet Workbench, as in [2.1.2 Java Program Development](javascript:ContentByName('pg-servlet-develop')) and [2.1.4 Elements of a Java Servlet](javascript:ContentByName('pg-servlet-elements')).

**Coding the Servlet**

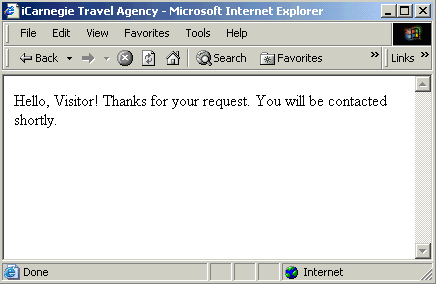
Preparing to write code for our first step, we use our knowledge from the HtmlWelcomeand PersonalWelcomeexamples in [2.1.4 Elements of a Java Servlet](javascript:ContentByName('pg-servlet-elements')), adding the following code:

* The out.printlnstatements that output the confirmation message including the dummy name *Visitor*. We write two statements, one that includes a greeting addressed to the user by name, and another for confirmation of the user's request.

Listing 2 shows our servlet code after we have finished editing for this step:

|  |  |
| --- | --- |
| 1:  2:  3:  4:  5:  6:  7:  8:  9: 10: 11: 12: 13: 14: 15: 16: 17: 18: 19: 20: 21: 22: 23: 24: 25: 26: 27: 28: 29: 30: 31: 32: 33: 34: 35: 36: 37: 38: 39: 40: 41: 42: 43: 44: 45: 46: 47: 48: 49: 50: 51: 52: 53: 54: 55: 56: 57: 58: 59: 60: 61: 62: 63: 64: 65: 66: 67: 68: 69: 70: 71: 72: 73: 74: 75: 76: 77: 78: 79: | /\*\*  \* Author: Enter your name here  \* Date: Enter the date here  \* Description: This servlet obtains user input from  \* TravelRequestForm, and responds with  \* a web page that has a confirmation message  \* with the user's name, and an image from  \* the travel destination.  \*  \*/  import java.io.\*;  import javax.servlet.\*;  import javax.servlet.http.\*;  public class TravelRequest extends HttpServlet {  public void doPost(  HttpServletRequest request,  HttpServletResponse response)  throws ServletException, IOException {  /\*\*  \* Indicate the content type (for example, text/html),  \* being returned by the response  \*/  response.setContentType("text/html");  /\*\*  \* Retrieve an output stream to use to send  \* data to the client  \*/  PrintWriter out = response.getWriter();  /\*\*  \* Get the user input from the form :  \* name and destination  \*/  /\* Add your code here \*/  /\*\*  \* Start by building the web page header  \*/  out.println(  "<!DOCTYPE HTML PUBLIC '-//W3C//DTD HTML 4.01 Transitional//EN'>");  out.println("<HTML>");  out.println("<HEAD>");  out.println("<TITLE>iCarnegie Travel Agency</TITLE>");  out.println("</HEAD>");  out.println("<BODY>");  /\*\*  \* Add the image  \*/  /\* Add your code here \*/  /\*\*  \* Add the confirmation message, with the name  \*/  out.println("Hello, Visitor!");  out.println(  "Thanks for your request. You will be contacted shortly.");  /\*\*  \* End by building the web page footer  \*/  out.println("</BODY>");  out.println("</HTML>");  } // end-doPost  } |
| **Listing 2** *TravelRequest.java* | |

Compile the servlet and run it using the form in *TravelRequestForm.html* and the workbench. The following screenshot shows a sample response of the servlet.



Preparing to write code for our second step, we know that the HTML code for including an image file in the Web page to be returned by the servlet would look something like:

<IMG src='/ny.jpg' alt='ny.jpg'>

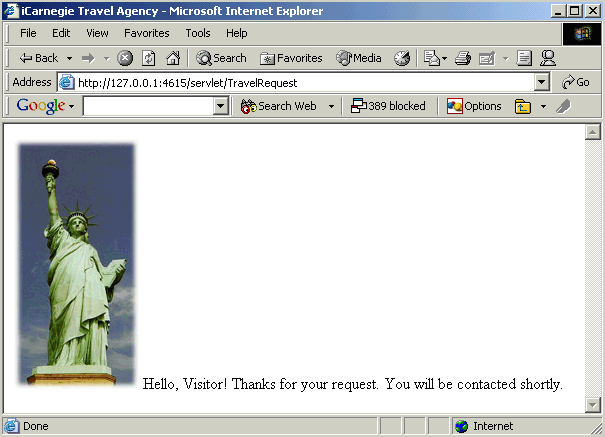
Notice that our specification does not tell us what the value of the alt attribute should be. Since we know that the HTML would not be valid without one, we just use the name of the image file we have for now. Following the HtmlWelcomeand PersonalWelcomeexamples in [2.1.4 Elements of a Java Servlet](javascript:ContentByName('pg-servlet-elements')), we add code for the following:

* The getParameterstatement for obtaining the value of the radio button input control called *Destination*
* The out.printlnstatements that output HTML code from the servlet. One of these statements includes the HTML code for the **IMG** tag whose **src** and **alt** attributes both specify the image file obtained earlier.

Listing 3 shows our servlet code after we have finished editing for this step:

|  |  |
| --- | --- |
| 1:  2:  3:  4:  5:  6:  7:  8:  9: 10: 11: 12: 13: 14: 15: 16: 17: 18: 19: 20: 21: 22: 23: 24: 25: 26: 27: 28: 29: 30: 31: 32: 33: 34: 35: 36: 37: 38: 39: 40: 41: 42: 43: 44: 45: 46: 47: 48: 49: 50: 51: 52: 53: 54: 55: 56: 57: 58: 59: 60: 61: 62: 63: 64: 65: 66: 67: 68: 69: 70: 71: 72: 73: 74: 75: 76: 77: 78: 79: 80: 81: | /\*\*  \* Author: Enter your name here  \* Date: Enter the date here  \* Description: This servlet obtains user input from  \* TravelRequestForm, and responds with  \* a web page that has a confirmation message  \* with the user's name, and an image from  \* the travel destination.  \*  \*/  import java.io.\*;  import javax.servlet.\*;  import javax.servlet.http.\*;  public class TravelRequest extends HttpServlet {  public void doPost(  HttpServletRequest request,  HttpServletResponse response)  throws ServletException, IOException {  /\*\*  \* Indicate the content type (for example, text/html),  \* being returned by the response  \*/  response.setContentType("text/html");  /\*\*  \* Retrieve an output stream to use to send  \* data to the client  \*/  PrintWriter out = response.getWriter();  /\*\*  \* Get the user input from the form :  \* name and destination  \*/  String destination = request.getParameter("Destination");  /\*\*  \* Start by building the web page header  \*/  out.println(  "<!DOCTYPE HTML PUBLIC '-//W3C//DTD HTML 4.01 Transitional//EN'>");  out.println("<HTML>");  out.println("<HEAD>");  out.println("<TITLE>iCarnegie Travel Agency</TITLE>");  out.println("</HEAD>");  out.println("<BODY>");  /\*\*  \* Add the image  \*/  out.println("<IMG src='/" + destination  + "' alt='" + destination + "'>");  /\*\*  \* Add the confirmation message, with the name  \*/  out.println("Hello, Visitor!");  out.println(  "Thanks for your request. You will be contacted shortly.");  /\*\*  \* End by building the web page footer  \*/  out.println("</BODY>");  out.println("</HTML>");  } // end-doPost  } |
| **Listing 3** *TravelRequest.java* | |

Compile the servlet, and run it using the form in *TravelRequestForm.html* and the workbench. Note that you must to remember to load the servlet class file *TravelRequest.class* **again** into the workbench, otherwise the workbench would simply use the servlet class file already loaded from the first step. The following screenshot shows a sample response of the servlet.



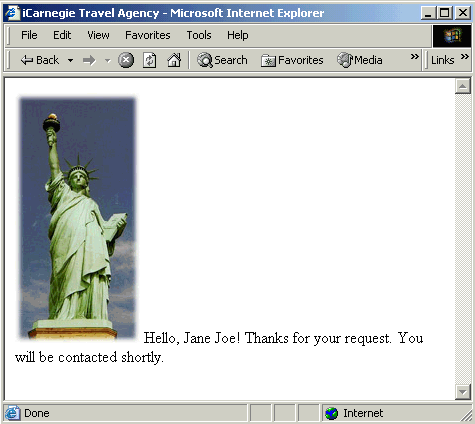
Preparing to write code for our third step, once again we use our knowledge [2.1.4 Elements of a Java Servlet](javascript:ContentByName('pg-servlet-elements')), and modify the code as follows:

* Add the getParameter statement for obtaining the value of the text input control called *Name*
* Edit the out.printlnstatement that outputs the greeting in the confirmation message to include the value obtained above instead of the dummy name *Visitor*.

Listing 4 shows our servlet code after we have finished editing for this step:

|  |  |
| --- | --- |
| 1:  2:  3:  4:  5:  6:  7:  8:  9: 10: 11: 12: 13: 14: 15: 16: 17: 18: 19: 20: 21: 22: 23: 24: 25: 26: 27: 28: 29: 30: 31: 32: 33: 34: 35: 36: 37: 38: 39: 40: 41: 42: 43: 44: 45: 46: 47: 48: 49: 50: 51: 52: 53: 54: 55: 56: 57: 58: 59: 60: 61: 62: 63: 64: 65: 66: 67: 68: 69: 70: 71: 72: 73: 74: 75: 76: 77: 78: 79: 80: 81: 82: | /\*\*  \* Author: Enter your name here  \* Date: Enter the date here  \* Description: This servlet obtains user input from  \* TravelRequestForm, and responds with  \* a web page that has a confirmation message  \* with the user's name, and an image from  \* the travel destination.  \*  \*/  import java.io.\*;  import javax.servlet.\*;  import javax.servlet.http.\*;  public class TravelRequest extends HttpServlet {  public void doPost(  HttpServletRequest request,  HttpServletResponse response)  throws ServletException, IOException {  /\*\*  \* Indicate the content type (for example, text/html),  \* being returned by the response  \*/  response.setContentType("text/html");  /\*\*  \* Retrieve an output stream to use to send  \* data to the client  \*/  PrintWriter out = response.getWriter();  /\*\*  \* Get the user input from the form :  \* name and destination  \*/  String destination = request.getParameter("Destination");  String name = request.getParameter("Name");  /\*\*  \* Start by building the web page header  \*/  out.println(  "<!DOCTYPE HTML PUBLIC '-//W3C//DTD HTML 4.01 Transitional//EN'>");  out.println("<HTML>");  out.println("<HEAD>");  out.println("<TITLE>iCarnegie Travel Agency</TITLE>");  out.println("</HEAD>");  out.println("<BODY>");  /\*\*  \* Add the image  \*/  out.println("<IMG src='/" + destination  + "' alt='" + destination + "'>");  /\*\*  \* Add the confirmation message, with the name  \*/  out.println("Hello, " + name + "!");  out.println(  "Thanks for your request. You will be contacted shortly.");  /\*\*  \* End by building the web page footer  \*/  out.println("</BODY>");  out.println("</HTML>");  } // end-doPost  } |
| **Listing 4** *TravelRequest.java* | |

Once again, compile the servlet, and run it using the form in *TravelRequestForm.html* and the workbench. Further, remember to load the servlet class file *TravelRequest.class* **again** into the workbench, to avoid the workbench using the servlet class file already loaded from the second step. The following screenshot shows a sample response of the servlet.



**Testing and Evaluating the Servlet**

As we are almost ready to declare our task as complete, we realize that we still need to test and evaluate it for conformance to the specification given to us. Our incremental approach to coding already forced us to adopt an incremental approach to testing as well. But, we do need to check that the response generated by our servlet is indeed as specified to us. After we have successfully completed the evaluation, we also need to make sure that the Web page constructed by the servlet is valid HTML. To validate the Web page generated by the servlet in response to a typical request, we save the Web page to a file such as *Test.html*. Then, we make sure that *Test.html* validates against the WDG HTML Validator as we have seen in [1.3 Introduction to HTML Forms and Servlets](javascript:ContentByName('pg-forms-servlets')). Note that in the event of any HTML errors flagged by the Validator, the out.printlnstatements in the servlet source file may need to be edited. Changes to the servlet source file would then need compilation and testing as we have done at the end of every step above, followed by re-validation.

When checking our servlet response against the sample servlet response we were given, we find that the only difference is in the placement of the confirmation message: the sample response shows the message after the image, while our response shows the message next to the image. While the servlet specification did not include any information on where the message should be laid out in relation to the image, here is something you can try. How about extending the code from Listing 3 to produce the correct placement of the message? Here is a hint: if you decide to make your servlet response match the sample response, you need one or two more out.printlnstatements in your servlet. Remember to follow the steps illustrated in this section:

* Define the problem you are attempting to solve. Here, you are trying to make the response of the TravelRequestservlet look identical to the sample response given. Essentially: you need to move the confirmation message on the page.
* Make a plan on how you can do this. This needs you to think about how you can solve the problem you have defined above, and how you will approach its solution. Note that you may need to split this up into smaller increments, each of which can be individually tested.
* Code the solution as per your plan above. Note that at the end of each step, you will need to compile, and run the servlet using the iCarnegie Servlet Workbench.
* Evaluate your solution against the specification given to you. This step includes the validation of the response Web page. Note that you may need to re-visit the previous step(s) of coding and testing in the event of any validation errors or any changes required to make your solution conform to the specification given.

**2.1.6 Guidelines for Java Development**

In this unit, we have looked at an object and a few different types of servlets. We have also learned how to develop a Java program. In this section, we will look at a few guidelines that help us develop programs more efficiently. Not only do they bring rewards in terms of optimizing the programming effort and time, they actually lead us to the goals of object-oriented programming with Java.

**The Process of Programming**

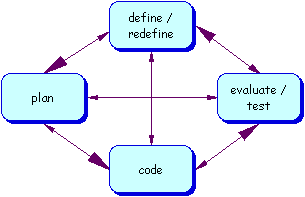
In the first unit of this course, a four-step process for developing Web pages was described to you. Those steps were:

1. Defining Web page content
2. Planning the look of the page and the needed links
3. Implementing the Web page by writing it incrementally in small steps
4. Evaluating the Web page in two ways: testing to see if the HTML is correct and looking to see if it meets the spec

This process is, in essence, the same process that programmers use to write a program. Later in the course, we will get into writing complex programs from scratch, when the process of programming needs particular attention. While the overhead of a non-existent or poor process is trivial when writing smaller programs, it becomes significant as the complexity of programs increases. If you re-visit [2.1.5 Planning Servlet Development](javascript:ContentByName('pg-servlet-plan');), you can deduce that we actually applied a slightly modified Web creation process there. Each component in this modified process can be called a phase instead of a step, where each phase could call for one or more iterations of the major step(s) that constitutes the phase. The revised process as applied to programming now looks like:

* Definition Phase: Define and/or redefine the problem
* Planning Phase: Plan a solution to the problem
* Coding Phase: Code the solution
* Evaluation Phase: Evaluate and test everything

The overall model is actually circular instead of linear; that is why the numbers disappeared. Here is a picture of the model:



This process is applied by the programmer to solve a programming problem. Take a guess where most of the time is spent. Do not look ahead at the next line!

The answer is in evaluation and testing, but we're getting ahead of ourselves. You have to start by defining the problem in your own words: you have to make it your own and for most of us that means restating it in our own words.  You often redefine the problem as part of doing so. You may have to ask questions to further your understanding of the problem. Once you have defined the problem, you have to evaluate your restatement to be sure that you are solving the correct problem. Don't laugh—solving the wrong problem is a common occurrence in a programming class.

If your evaluation shows that your restatement or redefinition is correct, you must begin to plan your solution (your program). This is where problem-solving skills come into the programming process. Most programmers solve a very small part of the original problem first, and then they solve the next small piece. This process is called divide- and-conquer. If you divide the problem into little pieces, a big problem can often be reduced to a series of smaller and simpler problems, each of which may be easier to code in Java.

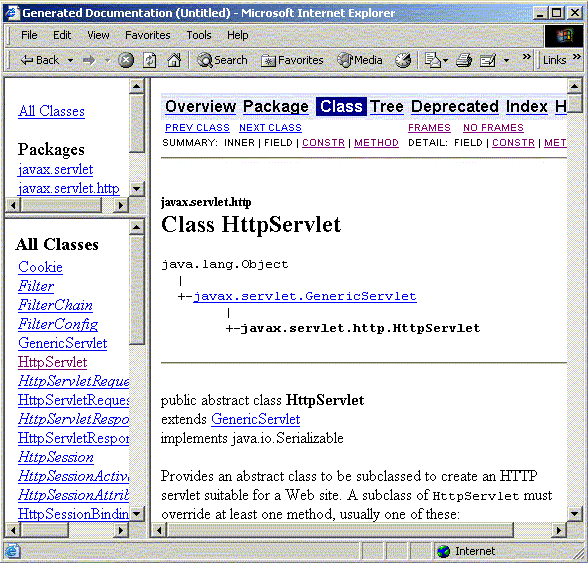
Once you have decided which piece of the problem to solve, you evaluate your decision and plan out your Java code for the simpler problem. After writing the code, it must be tested and inspected for errors. Once this test demonstrates that the code works as it is supposed to, you must return to the original problem with your working code for the piece you've solved, and begin the process again.

This may sound tedious and wasteful, but if applied in a reasonable manner, it will save you a great deal of time. Many novice programmers (and experienced ones, too) take a problem, go to the computer, and begin to code the solution immediately. While they may think that this is the fastest way to get the job done, our experience over the years is that the more time you spend on paper with this process, the less time you spend at the keyboard. **Programming is not coding. Coding is not programming.** The entire process as outlined above is programming. If you follow the process, you will actually find yourself spending as few hours coding at the keyboard as possible.

**The Java API Documentation**

As a Java programmer, you will need to use one or more of the Application Programming Interfaces (APIs) that Java offers. An API provides a set of classes and methods to a programmer, thus providing a re-usable foundation to build upon. As noted earlier in this section, a programmer cannot memorize everything about the programming language. A lot depends on practice. A crucial skill that can accelerate programming is the ability to find information fast and correctly by referencing the relevant documentation. All of this documentation is available at the [Java API Documentation](http://java.sun.com/j2se/1.4.2/docs/api/index.html" \t "external) site.

For example, classes and methods we used in the Welcome servlet can be looked up at the [Java Servlet Specification](http://java.sun.com/products/servlet/2.3/javadoc/index.html" \t "external) site. In the event that you wish to explore the HttpServlet class in more detail, you could find the information at [http://java.sun.com/products/servlet/2.3/javadoc/index.html](http://java.sun.com/products/servlet/2.3/javadoc/index.html" \t "external), as the following screenshot shows:



**Coding Style**

An important aspect of programming is developing a good style of writing code. This means that code written by you should be readable not just by you at some later time, but also by anyone else who may be expected to work on it, either to extend the functionality of the code or to fix any errors that may have been found. Most programmers adopt a specific set of rules or guidelines to be followed when writing code. Most of these depend on the programming language, but some do not. The [Java Coding Conventions](http://java.sun.com/docs/codeconv/html/CodeConvTOC.doc.html" \t "external) document presents an exhaustive set of such rules for Java. Most software development companies adopt these completely or else adopt a subset, depending upon what works well across their programming teams.

Another aspect of good programming style is writing code that is efficient, thus being able to function as expected in the most optimal way in terms of time and usage of system resources. Once again, this aspect can only get better with experience and thorough knowledge of the programming language.

**Commenting Source Code**

Documenting code is an important task for software developers. One element of documenting code correctly is including comments within the code. Well-commented code is easier to maintain because any developer who maintains it requires less time to understand the code's purpose and how the code works. This further reduces the software's total cost of ownership.

When you are commenting your class, you should assume that the reader of the comments will be a developer of average competence who knows nothing about the purpose of the class or how the class functions. Well-written comments should explain the purpose of the class somewhere near the beginning of the class. Each method should have at least one comment placed immediately before the method name. The comment should explain what the purpose of the method is, and how the method performs its function. Any complicated approach used to accomplish the method's function should be brought to the attention of the reader in order to minimize the time required by the developer to understand how the code functions.

When writing comments, it is important to express the intent behind the code. Comments that simply re-state what the code does are not helpful. At the same time, excessive commenting of source code must be avoided as it makes the code harder to read.

All Java source files you submit should be commented.  **Javadoc**

The Java SDK offers a tool for documenting code called Javadoc. If you type your comments in a specified format, you can run the Javadoc program on your class and an HTML document will automatically be generated that contains your comments in Java's standard documentation format.

For more information on how to use Javadoc, refer to [http://java.sun.com/j2se/javadoc/index.html](http://java.sun.com/j2se/javadoc/index.html" \t "external)

The SSD3 course requires the use of Javadoc for documenting your code.

**2.2 undamentals of Object-Oriented Programming**

In this module, we will learn some of the fundamental concepts of object-oriented programming, and examine the class definition for the Catfish class in more detail. We will also explore a class that models the attributes and behavior of HTML Web pages we are now familiar with.

### 2.2.1 Designing Classes

### Objects

Objects are fundamental building blocks of Object-Oriented Programming (OOP). To understand objects in OOP, let us draw upon the analogy of objects in the "real world."

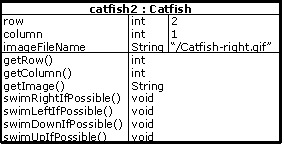
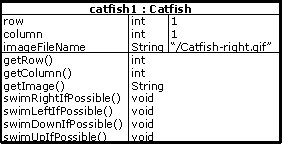
**Objects in the real world**: Objects in the real world are material or abstract things that humans can perceive. Here are some examples of physical objects at iCarnegie.

* iCarnegie employees
* The computers being used every day
* The coffeemaker

All the objects above occupy some physical space and are things that we can see, touch, and feel. In addition to physical objects, there are also abstract objects at iCarnegie. These objects do not occupy physical space, but we know they exist because we interact with them conceptually. The following are some examples:

* The iCarnegie trademark
* Each of ten courses
* iCarnegie business plan

**Software objects in a program**: An object in a program represents some real-world object, be it physical or conceptual. An object encapsulates data (state) and the mechanisms (behavior) that operate on the data. For example, here is a pictorial representation of the two catfish objects in our simple catfish demo:



Each catfish object has its own data: row, column, imageFileName. The operations on the data are getRow(), getColumn(), getImage(), swimRightIfPossible(), swimLeftIfPossible(), swimUpIfPossible(), swimDownIfPossible(). Notice that the two fish are located in different rows.

### Attributes, Values, and State

The values for all attributes of the fish object taken collectively define the state of the object. The computer stores these values as data. The state of an object can change over time. For example, when catfish1 swims right, the value for its column attribute changes, and hence the catfish object's state changes:

|  |  |  |
| --- | --- | --- |
|  |  |  |

### Behaviors, Operations, and Services

Objects can perform certain operations. Typically, an operation acts on data. In OOP, an object's operations are also called its behaviors. Following are some of Catfish's behaviors and the data manipulated by each behavior.

* swimRightIfPossible. Increments the value of column attribute, if possible. Changes the value of the imageFileName attribute
* swimUpIfPossible. Increments the value of row attribute, if possible. Changes the value of the imageFileName attribute.

In programming, behaviors include operations that query the state as well as operations that change the object state. Here are some examples of a fish object’s operations that query its state:

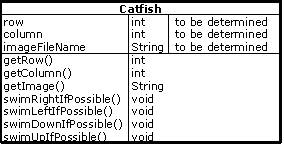
* getRow()
* getColumn()
* getImage()

The query operations above are not very useful to the fish object itself but are useful for any object that tries to display the fish. In other words, the fish object provides these query operations as a service to other objects. In OOP terminology, the services an object can provide consist of the set of operations it can perform. We will see later that an object can choose to restrict who can use its services.

### Class

A class is defined as "a software construct that defines the data (state) and methods (behavior) of the specific concrete objects that are subsequently constructed from that class" ([The Java Language Environment, Sun Microsystems](http://java.sun.com/docs/white/langenv/Object.doc2.html" \l "1372" \t "external)). A class is like a template that shapes all its object instances. All objects will have all the attributes and behaviors defined in the class. Conversely, an object cannot have any data or perform any operation that is absent in its class definition. For example, our simple Catfish class does not define a "eat" behavior, so none of the Catfish objects can perform an "eat" operation.

Here is the template defined by our simple Catfish:



The template is used to create objects. Each object has its own value for each attribute.

### Object State and the Object's Interface

#### Accessibility: public vs. private

As programmers, when we design a class, we have to consider carefully the behaviors and attributes that we make accessible to other objects. If we are too generous in giving access, unnecessary mistakes can occur. Let us illustrate this using a real world analogy. When car designers design a car, they are careful about the controls that they make available to drivers. The design of a car expects the driver to use the gas pedal, the brake pedal, and the steering wheel. The design does not expect the driver to drive by manipulating valves, gears, and pulleys from under the hood. Imagine the mistakes (and difficulties) that could occur if the driver could directly manipulate these. Although those items exist and are essential parts of the automobile, they can be manipulated only by other parts of the automobile.

When we design classes, we can think of things in a very similar way. By design, there are *private* aspects of a class that, much like the internals of a car, interact only with other parts of the car. In addition, there are *public* aspects of the objects available to the users of the object for interacting with the object -- for example, the steering wheel.

Objects of other classes can manipulate only the public attributes of objects and can use only the public services of an object. If you look at classes of objects from the outside, the only items that can be seen or manipulated are the public ones. We call the collection of all public aspects of a class its *interface*.  Java uses the keyword public to indicate such public aspects.

Information hiding is a term we use to describe that certain properties are inaccessible for public use. Java uses the keyword private to indicate such properties. It is important to note that Java allows us to specify not only data elements as private but also behaviors. In OOP, an object usually keeps its attributes private. This forces other objects to send a message requesting state change instead of directly manipulating the state.

For example, there are many turning wheels and moving belts in a car. If we were to model a car, it might be useful to capture these behaviors. For example, it might help us determine which of two engines operates best in a car. However, the only behavior to expose to the user is the gas pedal. Depressing the gas pedal, however, might initiate several behaviors inaccessible to the public, such as changing the oxygen-fuel mixture, allowing more fuel into the engine, or changing gears within the transmission. These behaviors, even though they are not accessible to the user of the class, might actually happen behind the scenes. We achieve this in programming by making such behaviors private.

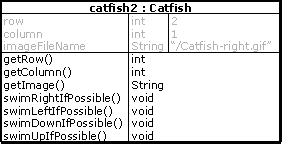
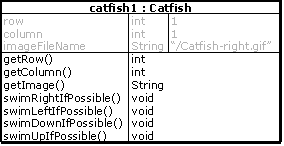
#### Object state

By keeping the internals of a car inaccessible from the user, we can prevent many mistakes. This ensures that the internal *state* of the car is correct. For example, the proportion of fuel and air mixed to produce combustion is always right.

The values of all aspects of an object constitute its *state.* The behaviors of an object, if properly written, will ensure that the state of an object is always *consistent.* This is called *state consistency.* For example, a properly operating car should never turn the tires left when the steering wheel turns right.

Consider the objects from classes available in the *Java* API. Programmers need to know quickly their purpose and proper usage. They are not usually concerned with their precise implementation. Such details in the documentation would distract programmers and complicate the programming task.

In our simple catfish example, we made the row, column and imageFileName attributes private. Therefore, they are inaccessible to objects of other classes, even though each catfish object has those attributes:



### Class Attributes

Most attributes have values specific to object. For example, each catfish instance has its own location values (i.e., row and column). Some attribute values, however, are common to all instances of a class. For example, let us assume that all catfish have to spend a certain amount of energy to swim to an adjacent cell. Since this amount does not depend on the state of any object, it is a property of the class as a whole. We call such properties class attributes. A class property does not belong to any single object of that class. Instead, it belongs to the class.

We can have a hundred catfish objects each with row and column values. In contrast, regardless of the number of catfish object instances, we need only one value in the program for the attribute that represents the energy to swim.

### Constants

The value for an attribute usually changes during the lifetime of the object, but there are some attributes for which the value does not change once it is set. For example, while the amount of fuel in a car changes during the lifetime of a car, the maximum capacity of the fuel tank does not change once the car is built. In other words, the "fuel tank capacity" attribute is a constant.

Just as an object can have constant attributes, a class can also have constant attributes. For example, the class attribute that represents the energy to swim is really a class constant, since the energy needed should not change once the program starts running.

### Designing a Class

The first step in designing a class is to identify all the object attributes and behaviors. Use the following heuristics:

* To identify attributes of an object, look for adjectives and possessive phrases such as

"the X of Y" and "Y's X" in the system specification.

"number of the account" and "client's name" are examples of an object's attributes. Usually, verbs are not attributes.

* To identify behaviors, look for verbs. For example, the statement

"the client deposits money into the account"

indicates the existence of the behavior "deposit."

Examine the other parts of the sentence to determine the sender and recipient of each message. The subject of the sentence above is "client." The subject initiates the action and hence is the sender of the message. The object is the entity who receives the message. In our sentence, there are two objects: "money," the direct object, and "account," the indirect object. To determine the recipient among the two, we use another OOP principle: To change the state of an object, send it a message rather than changing it directly. Therefore, we ask, "Whose state should be changed by the message?" In this case, the "Account" object's state should be changed. Hence, it becomes the recipient. The other object, "money," will be a parameter of the message.

While identifying the behaviors and attributes, some may be obvious and some not. To identify the non-obvious, inspect each operation and list the sequence of activities that constitute the operation and the data needed to perform the activities. For example, when a car is traveling, its speed is important. The same is true of its direction of travel. Moreover, depending on the type of program we are writing, it might also be important to know more details, such as the speed of the engine measured in RPMs. And, of course, we would not want to neglect one very important attribute -- the amount of fuel remaining in the car.

By carefully considering what objects do and what they need in order to do it, we can determine their behaviors, obvious attributes, and non-obvious attributes. We will illustrate this design process by designing two simple classes. After you clearly understand the process, practice it by designing your own classes for the application that follows the example.

### Design Process Example: Bank Teller

#### Introduction

Below is a simple applet that models a bank teller. It lets the user create accounts, select an account, deposit or withdraw money, and check the balance on accounts.

Side note: You are familiar with interactive GUI's of a different kind - HTML Forms. What we see below is an [applet](http://java.sun.com/applets/" \t "external). Since applets contain all the mechanisms needed to provide interactivity, they do not require the browser to send requests and receive responses from a web server for interactivity.

Please look at it, experiment with it a little bit, and become familiar with how it works. We are going to use it as an example to illustrate the design process for you.

#### Decomposing the Problem

The first step in designing a solution to this problem is to ask, "What are the components of this system?" In this case, we have three different components -- the one we can see and two we cannot. The Graphical User Interface (GUI for short) is the component we can see, and the underlying components we cannot see are BankAccount and AccountsLedger objects.

It is important to realize that these are separate objects. The GUI controls all user interactions and contains all that we see on the screen, including the text fields, the radio boxes, and the execute button. It does not actually do anything beyond manage the interaction with the user and request service from other objects.

One of the classes hiding behind the scenes, BankAccount, models a bank account – keeping track of the balance, handling withdrawals, handling deposits, and providing the balance when asked. There is one object of this kind for each bank account.

Another class hiding behind the scenes, AccountsLedger, models a ledger – maintaining a collection of accounts, creating and deleting accounts when asked, and looking up accounts.

In discussing the design of this applet, we are going to discuss both the AccountsLedger class and the BankAccount class separately. All we need to know about the GUI object are two things: It acts as an intermediary between the user and our BankAccount and AccountsLedger objects, and it utilizes the services of BankAccount objects to withdraw and deposit money, and the services of AccountsLedger object to look up specific accounts, and to create and delete accounts.

#### Designing the BankAccount Class

Let us begin by discussing the BankAccount class. This class defines the underlying object that models the bank account itself. A BankAccount object maintains a customer's account. A customer's account includes the customer name and account balance. The following are steps in designing such a class.

1. Identify the behaviors of object instances:

The first question we want to ask ourselves is "What services do object instances of this class provide?" In other words, what are the public *behaviors* (operations) of instances of this class? For each operation, we will also examine the sequence of activities that the object should perform. These activities will reveal the need for additional behavior and attributes, some of which may be public and some private.

Initially, three behaviors probably come to mind:

* + Deposit. Deposit a specified amount of money into the account. The amount of money is specified as an integer number. Integer implies that only whole dollar amounts are allowed i.e., no fractional amounts.
    - Sequence of activities needed to perform this operation: Increase the balance by the specified amount. Implication: We need to store the account balance as part of the object's state.

This operation should be part of the public interface of BankAccount since the objects of other classes should be able to send deposit messages to BankAccount objects. In the current design, the operation does not generate a reply. In an advanced design, BankAccount objects might generate a reply indicating an error if the deposit amount provided is a negative number.

* + Withdraw. Withdraw a specified amount of money from the balance. The amount of money is specified as an integer number with the integer representing whole dollar amounts.
    - Sequence of activities needed to perform this operation: Decrease the balance of the account by the specified amount of money. Implication: We need to store the account balance as part of the object's state.

This operation is also part of the public interface of BankAccount since objects of other classes should be able to send withdraw messages to BankAccount objects. In the current design, the operation does not generate a reply.

* + getBalance *(inquiry)*. Return the current balance of the account. The balance returned is an integer.

This behavior is also part of the public interface of BankAccount since the GUI object should be able to send getBalance messages to BankAccount objects and in response get the balance.

The above behaviors follow directly from the actions that the GUI exposes to the user. But, additionally, there is one other behavior of this class: the *constructor*. We know that the AccountsLedger will need to create a new instance of this class to model a specific bank account. In addition, this instance of the class will need to be initialized. This is generally true of objects. Therefore, we add one more behavior to our list:

* + *Constructor*. Create and initialize an account for the person whose name is specified as a String. Set the beginning balance to zero.

This constructor is also part of the public interface since the AccountsLedger object should be able to create new bank accounts. Note: All classes have public constructors with few exceptions. One of the exceptional situations is when, by design, we prohibit other classes from creating any object instances. This discussion is outside the scope of this course.

1. Identify the attributes of object instances:

We can approach this by asking the question, "What state should instances of this class maintain?" You might prefer to think of this as, "If the user of this object asks it to do something, what does it need to know in order to do it?"

In the case of the BankAccount class, each of the four methods manipulates the balance of the account. The *constructor* initializes it, withdraw and deposit modify it based on its old value, and getBalance returns its current value. With the exception of the constructor, the object needs to know the old value in order to behave properly. The BankAccountmust also remember the name of the account holder.

Therefore, we have two attributes:

* + **balance**. the total amount of money in the account represented as an integer number of dollars.

The value for this attribute can change during the lifetime of a BankAccount object and hence this is not a constant. Each object instance will have its own value for this attribute, so it is an instance attribute.

This attribute is not part of the public interface of the BankAccount class since we do not want objects of any other class to change the balance directly. By restricting access to withdraw and deposit messages, it becomes easier for a BankAccount object to maintain the integrity of its state.

* + **name**. the name of the account, a string of characters.

The value of this attribute cannot change during the lifetime of our BankAccount object once it has been created. Hence, this is a constant. Each object instance will have its own value for this attribute and hence it is an instance attribute and not a class attribute.

This attribute is part of the public interface of the BankAccount class since we do not care if other objects can see the name directly. We would be worried if other objects change the value of name without sending us a message. However, we know that other objects cannot successfully change the name since we have specified the name to be a constant.

1. Identify the class attributes of BankAccount

If we had decided to have a minimum balance for bank accounts, the "minimum balance" would be a class attribute as discussed earlier in the section "[Class Attributes](http://www.icarnegie.com/content/SSD/SSD1/2.1/normal/pg-intro-java-oop/pg-oop-fund/pg-design-class/pg-design-class.html" \l "class-attributes#class-attributes)." This is a class attribute, since the value of this attribute does not depend on the state of any object. The bank's policy applies to all bank accounts.

In this design, there is no minimum balance; hence, we do not have this attribute. If we choose to have this attribute, under what circumstances would you make this a constant? If this were not a constant, would you make this attribute public or private?

#### The Complete Specification for the BankAccount Class

Therefore, we have the complete English specification for the class as follows:

**Behaviors:**

* *Constructor*. Create an account for the person whose name is specified as a character string and set the beginning balance to $0.
* deposit. Increase the balance of the account by the specified amount of money. The amount of money is specified as an integer representing the dollars to be deposited. The behavior returns nothing.
* withdraw. Decrease the balance of the account by the specified amount of money. The amount of money is specified as an integer representing the dollars to be withdrawn. The behavior returns nothing.
* getBalance *(inquiry)*. Return the current balance of the account as an integer.

**Attributes:**

* balance*. the total amount of money in the account represented as an integer. This attribute is not a constant. This is an instance attribute (as opposed to a class attribute). This attribute is private.*
* name*. the customer name, a character string. This attribute is a constant, since the name does not change once it is set during account creation. This is an instance attribute. This attribute is public.*

### Design Process Example: AccountsLedger

An AccountsLedger object maintains a collection of bank accounts.

#### Designing the AccountsLedger Class

Let us perform a similar analysis to identify its behaviors and attributes:

1. Identify the behaviors of object instances:

As before, we want to ask ourselves the question, "What services does this object provide?"

Initially, three behaviors might come to mind:

* + *Create BankAccount*. Create a new BankAccount object for a customer whose name is specified as a character string. This behavior is part of the public interface since the GUI can send messages of this kind.
    - Sequence of activities to perform this operation: Add the newly created BankAccount object to the collection of bank accounts. Nothing needs to be returned as response to this message. Implication: We need a collection of BankAccount objects as an attribute. Let us call this attribute accounts.
  + Delete BankAccount. Delete a customer's BankAccount. The customer's name is specified as a String. This is part of the public interface since the GUI can send messages of this kind.
    - Sequence of activities: First, look for a BankAccount object whose name is equal to the specified String. If found, remove that object from the accounts collection. Nothing needs to be returned as response to this message. Implication: We need a method to lookup a bank account by name. Note: In this application, we delete an account even if the balance is not zero.
  + Lookup BankAccount. Find the BankAccount object that belongs to a customer. The customer's name is specified as a String. This is part of the public interface since the GUI can send messages of this kind.
    - Sequence of activities: Inspect each bank account and check if the account name matches the customer's name. If a match is found, return the account. The behavior returns a BankAccount object.
  + Constructor. Initialize a new AccountsLedger object.
    - Sequence of activities: The accounts attribute should be initialized to an empty collection.

1. Identify instance attributes:

The AccountsLedger object must maintain a collection of BankAccount objects that it can manipulate.

* + *accounts*. A collection of BankAccounts' in the ledger

1. Identify class attributes of AccountsLedger:

There isn't any attribute that applies to the AccountsLedger class as a whole.

#### The Complete Specification for the AccountsLedger Class

**Behaviors**:

* *Create*. Create a new BankAccount object for a customer whose name is specified. The name will be a String. Nothing needs to be returned as response to this message. This is part of the public interface.
* Delete. Delete a customer's bank account. The customer's name will be specified as a String. Nothing needs to be returned as response to this message. This is part of the public interface.
* Lookup. Lookup a bank account by customer name. The customer's name will be specified as a String. Returns a matching BankAccount object, if found. This is part of the public interface.
* Constructor. When a new AccountsLedger object is created, the accounts property should be initialized to an empty collection.

**Attributes**:

* *accounts*. A collection of BankAccounts maintained in the ledger.  
  For an AccountsLedger object to maintain a consistent state, we do not want other objects to change the value of this attribute without sending us a message. Hence, this attribute is not part of the public interface.

This completes our specification of BankAccount class and AccountsLedger class. We have to transform our English specification into Java. Before we do that, you should practice the design process.

### Student activity

After you have clearly understood the content above, complete the following activity. You will practice the following important skills:

* The ability to identify the properties and behaviors
* The ability to identify the appropriate accessibility criteria (private or public)
* The ability to distinguish between object attributes and class attributes
* The ability to identify constants

A phone book applet is given below. The phone book can be used to store and lookup contact information. When the button "add | modify" is pressed, if an entry with the specified name already exists in the phone book, then the phone number of that contact is modified. If such a contact does not exist, then a new contact with the specified name and number is added to the phone book. Notice that the phone number can include non-numeric characters as well. Please look at it, experiment with it, and become familiar with how it works.

Three classes make up this system -- one we can see and two we cannot. The GUI is the class we see and the two classes we cannot see are PhoneBook and Contact. PhoneBook has a list of contacts and Contact contains a name-phone pair.

Please produce an English specification of PhoneBook and Contact. You need not worry about the GUI. Your specification for each of these classes should be separate from one another and should describe both the behaviors and attributes of object instances and class attributes.

You should follow a format as similar as possible to the one we used in the example. To help you, please look below for a reminder of the questions you should ask along the way. Remember, you should write each of your two specifications separately, break each down into these three categories, and finally list and describe each item within each category.

Behaviors of object instances:

The first question to ask is, "What services do objects of this class provide?" In this case, we can ask the question, "What will the user of this class ask it to do?" List the sequence of activities needed to perform each behavior. The activity sequence could reveal additional behaviors and attributes that may not have been obvious. Do not forget the constructor.

Attributes of object instances:

The questions to ask are, "What attributes constitute the object state?" and "What instance attributes did you reveal when examining the behaviors?"

Class attributes:

You can approach this by asking the question, "Is there any attribute value that applies to the entire class?" In other words, is it enough to have only one value for the class, irrespective of the number of object instances? Does it make sense to have this value, even if there are no instances?

For each behavior, remember to answer the following

* Should the behavior be part of the public interface as discussed in [Accessibility: public vs. private](http://www.icarnegie.com/content/SSD/SSD1/2.1/normal/pg-intro-java-oop/pg-oop-fund/pg-design-class/pg-design-class.html" \l "public-vs-private#public-vs-private)? In other words, can objects of other classes directly send messages of this kind?
* To process this message, what should be specified with each message of this kind?
* What should be returned to the sender of this message?
* What sequence of activities is needed complete this operation?

For each attribute, remember to answer the following

* Is the attribute a constant as discussed in the section [Constants](http://www.icarnegie.com/content/SSD/SSD1/2.1/normal/pg-intro-java-oop/pg-oop-fund/pg-design-class/pg-design-class.html" \l "constants#constants)? In other words, does its value change once it is set?
* Should the attribute be part of the public interface as discussed in [Accessibility: public vs. private](http://www.icarnegie.com/content/SSD/SSD1/2.1/normal/pg-intro-java-oop/pg-oop-fund/pg-design-class/pg-design-class.html" \l "public-vs-private#public-vs-private)? In other words, if objects of other classes directly change the value of this attribute without sending a message to this object, can the object be expected to keep its state consistent at all times?

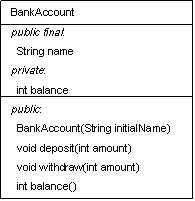
## 2.2.2 Transforming English Specification into Java

### Mapping English Terms to Java Terms

We now need to transform the English specification of our class design into Java. The following table gives us the required mapping.

|  |  |
| --- | --- |
| **Terms used in English specification** | **Technical term in Java** |
| attribute | field |
| object attribute | instance variable |
| class attribute | static variable (also known as static field) |
| constant attribute | final field |
| public | public |
| private | private |
| integer | int |
| string | String |
| collection of objects | Vector (Java has various kinds of collections. We limit ourselves to Vector in SSD1) |
| behavior | method |
| sending a message | invoking a method |
| specifications to be sent with messages | method parameters |
| constructor | constructor (constructor name should be identical to class name) |
| returns nothing | void |

With the above information, we pictorially represent our English specification as follows.



Notice the three parts in the diagram above: first is the class name, then the attributes and then the behaviors. The structure of a class definition in Java mimics the above structure:

Start class definition

definition of all fields

definition of constructors and methods

End class definition

### Java Rules and Conventions

In drawing the diagram above, we have followed Java rules and conventions. The compiler will enforce Java rules. Conventions, however, are recommendations. In SSD1, we follow the [Java conventions](http://java.sun.com/docs/codeconv/html/CodeConvTOC.doc.html" \t "externalWindow) published by Sun.

#### Class Name

Notice we have named our class BankAccount. There is no space between the two words. Java's rules state that a name cannot contain a space. Furthermore, according to Java conventions, [class names](http://java.sun.com/docs/codeconv/html/CodeConventions.doc8.html" \l "367" \t "external) should be nouns. Also according to convention, each word in the name should start with an upper case letter. The rest of the letters are lowercase.

#### Variables

We have two instance variables: balance and name.

* According to Java conventions, variable names start with lower case letters.
* Variable names should be meaningful relative to their purpose in the program.
* Notice that Java uses int for integer.

#### Constructor

* According to Java rules, constructors for a class must use the same name as the class. Hence, we name our constructor BankAccount.
* Java rules also specify that constructors should not return any values.

#### Method Names

Following Java conventions, we use verbs for method names. Example: withdraw. If a method name consists of more than one word, each word other than the first should start with an upper case letter. Example: getBalance

#### Parameter Specification

We specify the parameters for a method inside parentheses and right after the method name. If a method has no parameters, then the parentheses have nothing inside them, as in getBalance(). This is another rule in Java: parentheses should follow a method name. Parameters go inside the parentheses.

* Each parameter has a name.
* The type of each parameter goes before the name. Recall that the type specifies the range of possible values.
* Parameter names for constructors follow the pattern initialX where X is an instance variable. We follow this convention in SSD1.

#### Return Values

* The return value type is specified before the method name.
* When a method returns nothing, we use void. Example: void deposit(int amount)

### Mapping BankAccount to Java Code

Let us now transform our English specifications of BankAccount and AccountsLedger into equivalent class definitions in Java. A class definition starts by stating whether the class itself is publicly accessible to all other classes. All classes encountered in SSD1 will be public. Therefore, our BankAccount class definition starts and ends as given below. Notice the opening and closing curly braces denote the start and end of the class definition. Notice the documentation. Always remember to document your code. It greatly improves readability. Readability will be an important factor when grading your submissions.

|  |  |
| --- | --- |
| 1: 2: 3: 4: 5: 6: 7: 8: | /\*\*  \* This class implements a simple bank account.  \*  \* @author iCarnegie gk av  \* @version 1.1  \*/  public class BankAccount {  } |
| Listing 1  *BankAccount* class definition | |

Next, we add all the field definitions. Notice that each field starts by stating its accessibility: public or private. It is followed by the static and final keywords, if applicable, in that order. Then, the type of the field is specified: int or String for now. The field name follows the type. Recall that the type indicates the possible values that the attribute can take during the lifetime of the object. Each field definition ends with a semicolon.

|  |  |  |
| --- | --- | --- |
| 1:  2:  3:  4:  5:  6:  7:  8:  9: 10: 11: 12: 13: 14: | | /\*\*  \* This class implements a simple bank account.  \*  \* @author iCarnegie gk av  \* @version 1.1  \*/  public class BankAccount {  // Balance in dollars  private int balance;  // The name of the BankAccount - who does it belong to.  public final String name;  } |
| **Listing 2** *Definition of Class and Fields* | | |

Next, we add the constructor. Actions performed by the constructor should be specified within curly braces. Notice that our constructor currently does not perform any actions. We can fill in that detail later. Presently, we are just concerned with specifying the structure of the class. We will specify the name of the constructor and its parameters. Each parameter has a name and a type. In programming, the specification of a constructor's name and its parameters are called the constructor's signature.

|  |  |
| --- | --- |
| 1:  2:  3:  4:  5:  6:  7:  8:  9: 10: 11: 12: 13: 14: 15: 16: 17: 18: 19: 20: 21: 22: 23: | /\*\*  \* This class implements a simple bank account.  \*  \* @author iCarnegie gk av  \* @version 1.1  \*/  public class BankAccount {  // Balance in dollars  private int balance;  // The name of the BankAccount - who does it belong to.  public final String name;  /\*\*  \* Initializes the bank account's balance account  \* to $0 and sets its name  \*  \* @param initialName The name of the account.  \*/  public BankAccount(String initialName) {  }  } |
| Listing 3  Definition of Class, Fields and Constructor | |

Next, we add the other three methods. Just like the constructor above, our methods currently do not perform any actions. We can fill in that detail later. We will specify the name of each method, the return type, and parameters. In Java, a method's name and its list of parameters are called the method's signature.

|  |  |
| --- | --- |
| 1:  2:  3:  4:  5:  6:  7:  8:  9: 10: 11: 12: 13: 14: 15: 16: 17: 18: 19: 20: 21: 22: 23: 24: 25: 26: 27: 28: 29: 30: 31: 32: 33: 34: 35: 36: 37: 38: 39: 40: 41: 42: 43: 44: 45: 46: 47: 48: 49: 50: 51: | /\*\*  \* This class implements a simple bank account.  \*  \* @author iCarnegie gk av  \* @version 1.1  \*/  public class BankAccount {  // Balance in dollars  private int balance;  // The name of the BankAccount - who does it belong to.  public final String name;  /\*\*  \* Initializes the bank account's balance account  \* to $0 and sets its name  \*  \* @param initialName The name of the account.  \*/  public BankAccount(String initialName) {  }  /\*\*  \* Return the current balance of the account with  \* the amount of money specified as an integer number.  \*  \* @return the current balance of the account  \*/  public int getBalance() {  }  /\*\*  \* Increase the balance of the account by the specified  \* amount of money. The amount of money is  \* specified as an integer number.  \*  \* @param amount amount of money to add to the balance  \*/  public void deposit(int amount) {}  /\*\*  \* Decrease the balance of the account by  \* the specified amount of money. The amount of money is  \* specified as an integer number  \*  \* @param amount amount of money to subtract from the  \* balance.  \*/  public void withdraw(int amount) {}  } |
| Listing 4  Definition of Class, Fields, Constructor, and Methods | |

This almost completes our transformation. Before we claim completion, we need to make sure that all our punctuations are correct. For example, each opening brace should have a corresponding closing brace. The easiest mechanism to make sure we have a grammatically correct specification is to compile it. The compiler will say whether we have a grammatically incorrect specification. One of the fundamental tenets of programming is that a programmer makes sure the program compiles before handing it to someone else. Let us save this in a file named BankAccount.java and compile it by typing:

javac BankAccount.java

The compiler gives us the following error message:

[WIN2KLAP] Unit2/Module-2.2/skeleton> javac BankAccount.java

BankAccount.java:21: variable name might not have been initialized

public BankAccount(String initialName) {

^

BankAccount.java:30: missing return statement

public int getBalance() {

^

2 errors

The first error states that name is declared a constant, but is never initialized. The second error states that we have declared the method getBalance to return an integer, but it does not return anything. So, let us fix the first by initializing the BankAccount name and the second by returning an integer. Here is the fixed code:

|  |  |
| --- | --- |
| 1:  2:  3:  4:  5:  6:  7:  8:  9: 10: 11: 12: 13: 14: 15: 16: 17: 18: 19: 20: 21: 22: 23: 24: 25: 26: 27: 28: 29: 30: 31: 32: 33: 34: 35: 36: 37: 38: 39: 40: 41: 42: 43: 44: 45: 46: 47: 48: 49: 50: 51: 52: 53: | /\*\*  \* This class implements a simple bank account.  \*  \* @author iCarnegie gk av  \* @version 1.1  \*/  public class BankAccount {  // Balance in dollars  private int balance;  // The name of the BankAccount - who does it belong to.  public final String name;  /\*\*  \* Initializes the bank account's balance account  \* to $0 and sets its name  \*  \* @param initialName The name of the account.  \*/  public BankAccount(String initialName) {  name = initialName;  }  /\*\*  \* Return the current balance of the account with  \* the amount of money specified as an integer number.  \*  \* @return the current balance of the account  \*/  public int getBalance() {  return balance;  }  /\*\*  \* Increase the balance of the account by the specified  \* amount of money. The amount of money is  \* specified as an integer number.  \*  \* @param amount amount of money to add to the balance  \*/  public void deposit(int amount) {}  /\*\*  \* Decrease the balance of the account by  \* the specified amount of money. The amount of money is  \* specified as an integer number  \*  \* @param amount amount of money to subtract from the  \* balance.  \*/  public void withdraw(int amount) {}  } |
| Listing 5  Compilable Definition of Class, Fields, Constructor, and Methods | |

Compile it again. You will no longer see any errors. That completes our transformation of BankAccount class design. However, the class implementation is not complete until we implement the methods. Next, we need to transform AccountsLedger. Our English specification of AccountsLedger mentioned that it maintains a collection of BankAccount objects: creates BankAccount objects, deletes them, and looks them up. AccountsLedger does this by holding on to a collection of references to the BankAccount objects in a Vector. Before we discuss AccountsLedger, we need to understand object creation, deletion, object references, and collections. We will discuss these in the next page. Before you proceed to the next page, we recommend that you complete the following student activity.

### Student Activity

After you have clearly understood the content above, complete the following activity. You will practice the following important skills:

* The ability to express attributes and behaviors in a concise pictorial representation
* The ability to transform an English statement of class structure into a compilable Java class definition

Draw a pictorial representation of your Contact class much like the representation given above. Then, write the equivalent Java code and compile it. Fix all errors reported by the compiler.

## 2.2.3 Lifecycle of Objects

### Creating Objects

The process of creating an object instance of a class is called instantiation. In Java, we use the new operator to create objects. For example, to create a new BankAccount object for Bob, we can write:

new BankAccount("Bob")

To use the newly created object, we need a mechanism to refer to it, which we will discuss next.

### Referencing Objects

A reference variable can hold a reference to an object. A reference variable is any variable in Java whose type is a class.

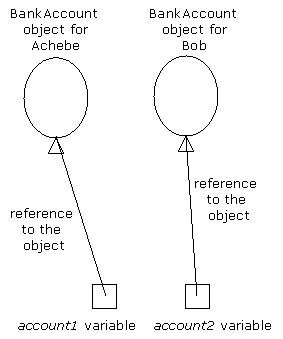
We will use an analogy to illustrate the relationship between an object, a reference to the object and a reference variable. This analogy is a modified version of the one followed in the book "Beginning Java Objects" by Jacquie Barker. If an object is analogous to a helium balloon, a reference is analogous to a thread that hangs from the balloon. A slot that holds onto the thread is analogous to a reference variable. Let us consider the case of two reference variables, account1 and account2, and two BankAccount objects, one corresponding to Achebe and one corresponding to Bob. Here is the code for account1 holding a reference to Achebe's BankAccount and account2 holding a reference to Bob's BankAccount object, and the corresponding pictorial representation:

BankAccount account1;

BankAccount account2;

account1 = new BankAccount("Achebe");

account2 = new BankAccount("Bob");



The new operator returns a reference to the newly created object instance. We stored the created references in reference variables. Java allows a short-form that combines the declaration of reference variables and object creation:

BankAccount account1 = new BankAccount("Achebe");

BankAccount account2 = new BankAccount("Bob");

### Garbage Collecting Objects and nulls

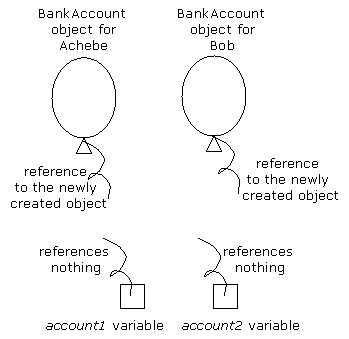
If the example in the previous subsection did not store the references, the reference variables will not refer to any object. A reference variable that refers to "nothing" is said to have a null value. Furthermore, there would be no way of using the newly created objects since they are unreachable. The unreachable objects will "fly away." In Java terminology, unreachable objects will be "garbage collected" by the JVM. Here is the code and the corresponding pictorial representation for inaccessible (unreachable) references:

BankAccount account1; // has null value

BankAccount account2; // has null value

new BankAccount("Achebe"); // reference is lost. Achebe is unreachable!

new BankAccount("Bob"); // reference is lost. Bob is unreachable!



### Multiple references to an object

A number of reference variables can reference the same object. Here is code and the pictorial analogy when account1 and account2 reference Achebe's account:

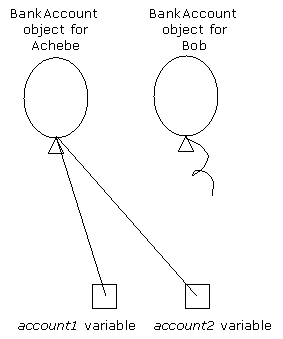
BankAccount account1; // has null value

BankAccount account2; // has null value

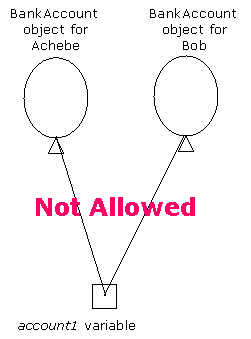
account1 = new BankAccount("Achebe"); // account1 has Achebe's reference

new BankAccount("Bob"); // Reference lost. Bob is unreachable!

account2 = account1; // account2 also has Achebe's reference



A variable can hold only one reference at a time:



The reference stored in a reference variable can change over time. When, the reference variable stores a new reference, the old reference is lost. Here is the code snippet and the pictorial analogy upon completing execution of the code snippet:

BankAccount account1; // has null value

BankAccount account2; // has null value

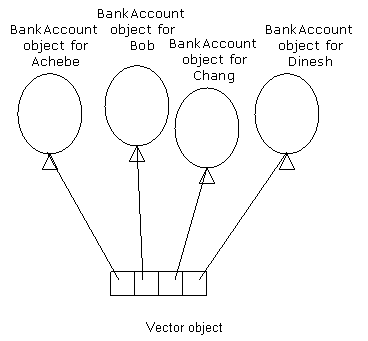
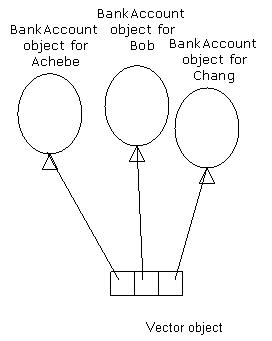
account1 = new BankAccount("Achebe");

account2 = new BankAccount("Bob");

account1 = account2; // Reference to Achebe is lost.

### Vector

We mentioned earlier that our AccountsLedger class maintains a collection of BankAccount objects. Java provides various kinds of collections. The one we use in SSD1 is Vector. Other kinds of collections are outside the scope of SSD1. We will briefly discuss Vector here. A Vector object can hold a collection of object references. Here is a pictorial representation of a Vector object that holds three BankAccount objects:



#### Vector grows

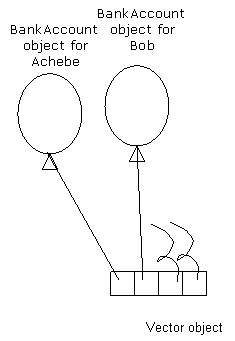
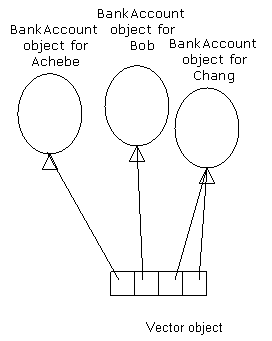
A Vector can grow to accommodate additional object references. When we add a new BankAccount object for Dinesh, our Vector object looks like this:

#### Vector is an ordered collection

The references in a Vector are ordered. The first element in the Vector holds a link to the BankAccount object for Achebe, the second to Bob, the third to Chang and the fourth to Dinesh. The starting index is zero. Therefore, the last element will have an index of size - 1.

#### Vector can hold duplicates

A Vector can hold duplicate references. Here is an example of our vector holding duplicate references:



#### Vector can hold null values

A Vector can hold references to nothing:

Notice the following points about the diagrams:

* We are showing the four BankAccount objects outside Vector. A Vector holds only references to the objects. In other words, the Vector does not contain the object itself.
* Each slot in the Vector acts like a reference variable. Each slot can hold only one reference.

In Java, the Vector class is defined in the java.util package. Any class that uses it, for example our AccountsLedger class, has to import it (recall the discussion of import from [2.1 Programming in Java](javascript:ContentByName('pg-prog-java');)).

#### A Vector object is like any other object!

A reference variable can be declared to hold a Vector reference:

Vector accounts; // has null value.

For the reference variable to hold a reference to an actual Vector object, we have to create a new Vector object:

Vector accounts = new Vector();

If a Vector object has no references pointing to it, the Vector object becomes unreachable. Such a Vector object will be garbage collected. If the objects referred to by the Vector have no other references pointing to them, they will also be garbage collected:

Vector accounts = new Vector();

BankAccount tempAccount = new BankAccount("Achebe");

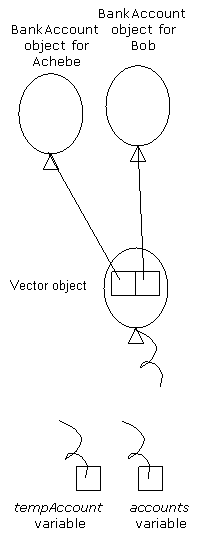
accounts.add(tempAccount);

tempAccount = new BankAccount("Bob");

accounts.add(tempAccount);

accounts = null;

tempAccount = null;



### Student Activity: Vector

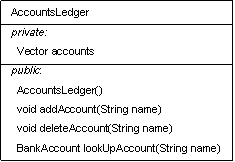
Write code to accomplish the following features.

1. Modify the code above so you have another reference variable named accounts2 that has a reference to the Vector object.
2. Add a BankAccount object for "Barathi" to the Vector object.
3. If your code for feature #2 (add "Barathi" account to Vector) came after the code for feature #1, would accounts2 point to the modified Vector object that has a Barathi reference?

This concludes our brief introduction to Vector. Let us turn our attention back to the AccountsLedger class.

### Transforming AccountsLedger class

Here is a pictorial representation of AccountsLedger:



Note the following:

* The instance variable accounts is a Vector.
* The constructor is named AccountsLedger, the same name as the class.
* The lookUpAccount method returns a BankAccount object.

Let us now transform our English specification into Java code as follows:

|  |  |
| --- | --- |
| 1:  2:  3:  4:  5:  6:  7:  8:  9: 10: 11: 12: 13: 14: 15: 16: 17: 18: 19: 20: 21: 22: 23: 24: 25: 26: 27: 28: 29: 30: 31: 32: 33: 34: 35: 36: 37: 38: 39: 40: 41: 42: 43: | import java.util.Vector;  /\*\*  \* This class operates multiple bank accounts  \*  \* @author iCarnegie gk, ar, av  \* @version 1.1  \*/  public class AccountsLedger {  /\* Vector of different bank accounts \*/  private Vector accounts;  /\*\*  \* Initializes a new teller  \*/  public AccountsLedger() {}  /\*\*  \* Adds a new account, by name to the teller with a $0 balance  \*  \* @param name the name of the account, e.g. "Bob"  \*/  public void addAccount (String name) {}  /\*\*  \* Delete an existing account  \*  \* @param name the name of the account, e.g. "Bob"  \*/  public void deleteAccount (String name) {}  /\*\*  \* Find the BankAccount whose customer name is specified.  \*  \* @param name - customer name  \* @return <code>BankAccount</code> object corresponding to a customer, if found.  \* <code>null</code>, otherwise.  \*/  public BankAccount lookupAccount(String name) {  }  } |
| Listing 1  *AccountsLedger* class definition | |

As before, let us compile our code to make sure it is grammatically correct. The compiler gives us the following error message:

[WIN2KLAP] big-revision/ssd1/jnk> javac AccountsLedger.java

AccountsLedger.java:41: missing return statement

public BankAccount lookupAccount(String name) {

^

1 error

The error states that the lookUpAccount method has been defined to return a BankAccount object. However, the method currently does not return anything. To fix this error, let us return a null reference:

|  |  |
| --- | --- |
| 1:  2:  3:  4:  5:  6:  7:  8:  9: 10: 11: 12: 13: 14: 15: 16: 17: 18: 19: 20: 21: 22: 23: 24: 25: 26: 27: 28: 29: 30: 31: 32: 33: 34: 35: 36: 37: 38: 39: 40: 41: 42: 43: 44: | import java.util.Vector;  /\*\*  \* This class operates multiple bank accounts  \*  \* @author iCarnegie gk, ar, av  \* @version 1.1  \*/  public class AccountsLedger {  /\* Vector of different bank accounts \*/  private Vector accounts;  /\*\*  \* Initializes a new teller  \*/  public AccountsLedger() {}  /\*\*  \* Adds a new account, by name to the teller with a $0 balance  \*  \* @param name the name of the account, e.g. "Bob"  \*/  public void addAccount (String name) {}  /\*\*  \* Delete an existing account  \*  \* @param name the name of the account, e.g. "Bob"  \*/  public void deleteAccount (String name) {}  /\*\*  \* Find the BankAccount whose customer name is specified.  \*  \* @param name - customer name  \* @return <code>BankAccount</code> object corresponding to a customer, if found.  \* <code>null</code>, otherwise.  \*/  public BankAccount lookupAccount(String name) {  return null;  }  } |
| Listing 2  Compilable *AccountsLedger* class definition | |

Compiling it again shows that we have no errors. That concludes our transformation of AccountsLedger design. However, we are not done with our implementation until we implement all the methods.

### Student Activity: Transform PhoneBook

After you have clearly understood the content above, draw a pictorial representation of your PhoneBook class. Then, write the equivalent Java code and compile it. Fix all errors reported by the compiler.

**2.2.4 The HtmlPage Class**

Given that nearly all servlets we develop in this course will generate a Web page in response, the following observations deserve some thought:

* As can be seen from the source code for the HtmlWelcome servlet in [2.1.4 Elements of a Java Servlet](javascript:ContentByName('pg-servlet-elements');) and the TravelRequest servlet in [2.1.5 Planning Servlet Development](javascript:ContentByName('pg-servlet-plan');), some part of Web page generation is common to all servlets.
* We need one out.println statement per line of HTML code in the servlet source code. This indicates that, as the response Web pages get larger or more complex, our servlet code will need many repetitive out.println statements.
* It would be useful to design a class that models the attributes and behavior of a Web page. Servlets in our programs could send messages to an object of this class to handle the task of HTML code generation, thus relieving the servlets of that task to some extent. This object would in essence encapsulate part of the HTML generation functionality.
* Using a class to construct the Web page helps us localize HTML code generation to one class, which makes the task of HTML validation a lot easier. If we make sure that the HTML code generated by this class is valid, we need not worry about validation errors in that code.

In this section, we will explore the design for a class, HtmlPage, that models the type of Web pages we worked with in Unit 1.We will then modify the HtmlWelcome servlet from [2.1.4 Elements of a Java Servlet](javascript:ContentByName('pg-servlet-elements');) to send messages to an instance of this class for building the response Web page.

**Designing the HtmlPage Class**

Along the lines of the design process followed in [2.2.1 Designing Classes](javascript:ContentByName('pg-design-class');), let us give some thought to the attributes and behaviors that must be defined by the HtmlPage class.

The most important service provided by this class is the construction of a Web page with valid HTML code. This must include the following:

* A header built with tags: <HTML>,<HEAD>,</HEAD>,<TITLE> and </TITLE>. Note that the title of the Web page, if specified, must be enclosed between the <TITLE> and </TITLE> tags. Further, the <TITLE> and </TITLE> tags must be placed within the <HEAD> and </HEAD> tags themselves.
* A footer built with tag: </HTML>
* A body enclosed between the header and the footer. The body starts with the tag <BODY> and includes all HTML code between the <BODY> and </BODY>tags. This could include hyperlinks, images, or simply lines of text. For now, let's limit the class design to including text in the body of the Web page.

We also need to consider the following about the Web pages constructed by this class:

* The Web page must include the <DOCTYPE> element as follows:

<!DOCTYPE HTML PUBLIC '-//W3C//DTD HTML 4.01 Transitional//EN'>

* Users may not always specify a title for the Web page.
* Users may specify a background image or a background color for the Web page. Note that some users may specify both, while others may specify neither. The Web page must be built with the appropriate attributes for the <BODY> tag.

**English Specification for the HtmlPage Class**

Having given some thought to how objects of HtmlPage class must behave, we can summarize the English specification for this class as follows:

* Behaviors of object instances:
  + *Constructor*. Create an HtmlPage object. Set the initial values of all attributes to be empty strings.
  + *getHeader*. Build the HTML code for the Web page header. Returns a String object.
  + *getFooter*. Build the HTML code for the Web page footer. Returns a String object.
  + *setTitle*. Set the value of the attribute that models the Web page title as specified. Title specified as a String representing the desired Web page title. Returns nothing.
  + *setBackgroundImage*. Set the value of the attribute that indicates the background image for the Web page as specified. Background image specified as a String representing the name of the image file. Returns nothing.
  + *setBackgroundColor*. Set the value of the attribute that indicates the background color for the Web page as specified. Background color specified as a String representing the color taken by the BGCOLOR attribute of the <BODY> tag. Returns nothing.
  + *getBody*. Build the HTML code for the body of the Web page. Returns a String object. Includes the following sequence of activities:
    - Start with the <BODY> tag, including the **background** and **bgcolor** attributes from the corresponding attributes for the Web page.
    - Include the HTML code for the body of the Web page.
    - End with the </BODY> tag.
  + *getDoctype*. Build the HTML code for the <DOCTYPE> element to be included at the start of the Web page. Returns a String object.
  + *addText*. Modify the value of the attribute that models the body of the Web page by adding a line of text. Returns nothing.
  + *buildHtml*. Complete the construction of the Web page. Returns a String object. Includes the following sequence of activities:
    - Start with the HTML code for the <DOCTYPE> element returned by the getDoctype behavior
    - Include the HTML code for the header returned by the getHeader behavior
    - Include the HTML code for the body returned by the getBody behavior
    - Include the HTML code for the footer returned by the getFooter behavior
* Attributes of object instances:
  + *pageTitle*. Represents the Web page title. Stored as a String.
  + *pageBody*. Represents the body of the Web page. Stored as a String.
  + *bgImage*. Represents the name of the image file to be used as background image for the Web page.  
    Stored as a String.
  + *bgColor*. Represents the background color for the Web page. Stored as a String.

**Mapping English Specification to Java Code**

Try to map the Java code the class definition of HtmlPage class in Listing 1 to the English specification for the HtmlPage class. The following observations will help you along:

* The class has four String attributes. All attributes are private.
* The behaviors getDoctype, getHeader, getFooter, and getBody are part of the private interface.
* The behaviors setTitle, setBackgroundImage, setBackgroundColor, addText, and buildHtml are part of the public interface.
* The constructor HtmlPagetakes no arguments. It sets the initial values of all attributes to be empty strings.

|  |  |
| --- | --- |
| 1:  2:  3:  4:  5:  6:  7:  8:  9: 10: 11: 12: 13: 14: 15: 16: 17: 18: 19: 20: 21: 22: 23: 24: 25: 26: 27: 28: 29: 30: 31: 32: 33: 34: 35: 36: 37: 38: 39: 40: 41: 42: 43: 44: 45: 46: 47: 48: 49: 50: 51: 52: 53: 54: 55: 56: 57: 58: 59: 60: 61: 62: 63: 64: 65: 66: 67: 68: 69: 70: 71: 72: 73: 74: 75: 76: 77: 78: 79: 80: 81: 82: 83: 84: 85: 86: 87: 88: 89: 90: 91: 92: 93: 94: 95: 96: 97: 98: 99: | import java.io.\*;  public class HtmlPage {  private String pageTitle;  private String pageBody;  private String bgImage;  private String bgColor;  /\*\*  \* Constructor  \*/  HtmlPage() {  pageTitle = "";  pageBody = "";  bgImage = "";  bgColor = "";  }  /\*\*  \* Build the page header  \*/  private String getHeader() {  return ("<HTML><HEAD><TITLE>" + pageTitle +  "</TITLE></HEAD>");  }  /\*\*  \* Build the page footer  \*/  private String getFooter() {  return ("</HTML>");  }  /\*\*  \* Set the page title  \*/  public void setTitle(String pTitle) {  pageTitle = pTitle;  }  /\*\*  \* Set the body background image attribute  \*/  public void setBackgroundImage(String imageName) {  bgImage = imageName;  }  /\*\*  \* Set the body background color attribute  \*/  public void setBackgroundColor(String colorValue) {  bgColor = colorValue;  }  /\*\*  \* Build the page body  \*/  private String getBody() {  return ("<BODY background='" + bgImage + "' bgcolor='"  + bgColor + "'>"  + pageBody  + "</BODY>");  }  /\*\*  \* Build the document type  \*/  private String getDoctype() {  return (  "<!DOCTYPE HTML PUBLIC '-//W3C//DTD HTML 4.01 Transitional//EN'>");  }  /\*\*  \* Add a line of text to the body  \*/  public void addText(String textString) {  pageBody = pageBody + textString;  }  /\*\*  \* Build the page now  \*/  public String buildHtml() {  String pageString = getDoctype() + // Start with document type  getHeader() + // Add header  getBody() + // Add body  getFooter(); // Add footer  return pageString;  }  } |
| **Listing 1** *HtmlPage.java* | |

See if you can you answer the following questions about the design and implementation of HtmlPage class:

* How does the class design ensure that the Web page generated would always have the right structure consisting of the <DOCTYPE> element, a header, a footer, a body, and a footer?
* Why are all of the attributes pageTitle, pageBody, bgColor, and bgImage private?
* Why are the behaviors getDoctype, getHeader, getFooter, and getBody private?
* Suppose you decide to use the HtmlPage class in your servlet to generate a Web page. You therefore create an instance of the HtmlPage class and send the buildHtmlmessage to that instance. Would you still get a valid Web page?
* Suppose you need to add a line of italicized or bold text to the body of the Web page. Can you do this with the present design and not write any HTML code yourself? This should give some ideas about how you can extend the behaviors provided by this class.

**Building Response Web Page with HtmlPage Class**

Listing 2 shows the HtmlWelcome servlet from [2.1.4 Elements of a Java Servlet](javascript:ContentByName('pg-servlet-elements');) that builds a response Web page to display a welcome message. Note that the servlet source code includes ten out.println statements on lines 25–35 to build an extremely simple Web page to display a single line of text. Note also that the servlet source code includes HTML code.

|  |  |  |
| --- | --- | --- |
| 1:  2:  3:  4:  5:  6:  7:  8:  9: 10: 11: 12: 13: 14: 15: 16: 17: 18: 19: 20: 21: 22: 23: 24: 25: 26: 27: 28: 29: 30: 31: 32: 33: 34: 35: 36: 37: | import java.io.\*;  import javax.servlet.\*;  import javax.servlet.http.\*;  public class HtmlWelcome extends HttpServlet {  public void doPost(HttpServletRequest request,  HttpServletResponse response)  throws ServletException, IOException {  /\*\*  \* Indicate the content type (for example, text/html),  \* being returned by the response  \*/  response.setContentType("text/html");  /\*\*  \* Retrieve an output stream to use to send  \* data to the client  \*/  PrintWriter out = response.getWriter();  /\*\*  \* Write the response  \*/  out.println(  "<!DOCTYPE HTML PUBLIC '-//W3C//DTD HTML 4.01 Transitional//EN'>");  out.println("<HTML>");  out.println("<HEAD>");  out.println("<TITLE>");  out.println("Welcome");  out.println("</TITLE>");  out.println("<BODY>");  out.println("Welcome to iCarnegie!");  out.println("</BODY>");  out.println("</HTML>");  }  } | |
| **Listing 2** *HtmlWelcome.java* | |

Listing 3 shows a modified HtmlWelcome servlet that uses an instance of the HtmlPage class to generate the response Web page. Note that the servlet source code includes just one out.println statement on line 36 to build the same Web page. Note also that the servlet source code includes no HTML code.

|  |  |
| --- | --- |
| 1:  2:  3:  4:  5:  6:  7:  8:  9: 10: 11: 12: 13: 14: 15: 16: 17: 18: 19: 20: 21: 22: 23: 24: 25: 26: 27: 28: 29: 30: 31: 32: 33: 34: 35: 36: 37: 38: | import java.io.\*;  import javax.servlet.\*;  import javax.servlet.http.\*;  public class HtmlWelcome extends HttpServlet {  public void doPost(HttpServletRequest request,  HttpServletResponse response)  throws ServletException, IOException {  /\*\*  \* Indicate the content type (for example, text/html),  \* being returned by the response  \*/  response.setContentType("text/html");  /\*\*  \* Retrieve an output stream to use to send  \* data to the client  \*/  PrintWriter out = response.getWriter();  /\*\*  \* Create an instance of the HtmlPage class  \*/  HtmlPage myPage = new HtmlPage();  /\*\*  \* Send appropriate messages to the HtmlPage object  \*/  myPage.setTitle("Welcome");  myPage.addText("Welcome to iCarnegie!");  /\*\*  \* Write the response  \*/  out.println( myPage.buildHtml() );  }  } |
| **Listing 3** *HtmlWelcome.java* | |

The modified HtmlWelcome servlet was obtained with the following changes:

* Create an object of the HtmlPage class using the new operator and the constructor for the HtmlPage class as seen on line 25.

HtmlPage myPage = new HtmlPage("");

The identifier myPage refers to the instance returned by the constructor.

* Send the setTitle message to myPage to add a title to the Web page as seen on line 30.

myPage.setTitle("Welcome");

* Send the addText message to myPage as seen on line 31. This adds HTML code to display the welcome message in the Web page.

myPage.addText("Welcome to iCarnegie!");

* The statement on line 36 involves two steps. First, send the buildHtml message to myPage. This completes the construction of the Web page and returns a String object to the servlet. Next, send the println message to the HttpServletResponse object response with the String object from the first step as its argument.

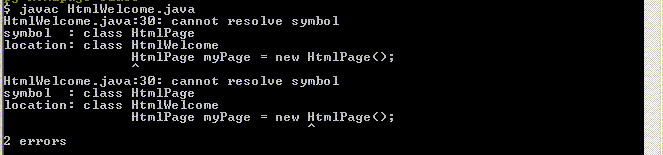
out.println(myPage.buildHtml());

**Compiling and Running Multiple Classes**

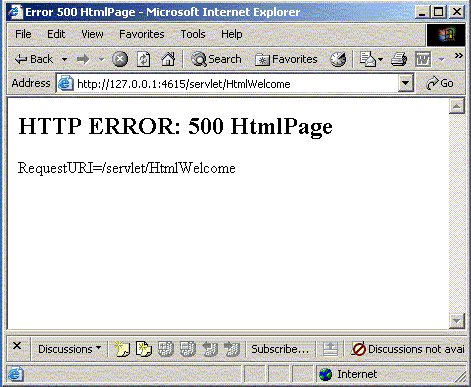
When we developed the HtmlWelcome servlet in [2.1.4 Elements of a Java Servlet](javascript:ContentByName('pg-servlet-elements');), we compiled only one source file. With the changes to the source for the HtmlWelcome servlet as shown in Listing 3, we must take additional steps to run the servlet. In addition to compiling the servlet source file, we need to compile the source file for the HtmlPage class. This can be done by specifying both source files when you use the *javac* command for compilation as shown below.



The following screenshot shows the compilation error you will see if you forget to specify the source file for the HtmlPage class when compiling the source file for HtmlWelcome servlet.



To run the HtmlWelcome servlet that uses the HtmlPage class, the procedure is the same as followed in [2.1.4 Elements of a Java Servlet](javascript:ContentByName('pg-servlet-elements');) except that now the class files for *both* the servlet and the HtmlPage class need to be loaded into the iCarnegie Servlet Workbench. If you submit the form in *HtmlWelcomeForm.html* without loading the file *HtmlPage.class* into the workbench, you will see the following error in your browser window:



The error message indicates that the Web server encountered an internal error, as the class file for HtmlPage class was not found.

The response Web page you see when you submit the form in *HtmlWelcomeForm.html* will not look any different from that seen in [2.1.4 Elements of a Java Servlet](javascript:ContentByName('pg-servlet-elements');). But, note that your Java code in *HtmlWelcome.java* included no HTML code. The HtmlWelcome servlet created and sent messages to an HtmlPage object to generate all HTML code. Besides keeping HTML code generation out of the servlet code and thus making it more readable, it helped avoid writing code that will be repeated in most servlets through the remainder of the course. In the process, you have learned how to compile and run a servlet that uses a class designed by you.

Here is a class design task you can try:

* Design a class HtmlAnchor that generates the HTML code for a hyperlink using the opening and closing anchor tags : <A> and </A>. The English specification for this class is as follows:

Behaviors:

* + *HtmlAnchor*. Constructor for the HtmlAnchor class. Creates an HtmlAnchor object. Sets the values of both the attributes to be as specified in the arguments.
  + *buildHtml*. Similar to the buildHtmlbehavior you have seen for the HtmlPage class. Sequence of activities to complete execution of this message includes building a String object representing the HTML code for the hyperlink.

Attributes of the object instances:

* + *displayText*. The text to be displayed for the hyperlink.
  + *targetLocation*. The target location for the hyperlink.

Note that the attributes are based on the information that must be specified when writing the HTML code for a hyperlink, as you may recall from [1.2.4 Creating Links to Other Documents](javascript:ContentByName('pg-links-to-othr-docs');). Both attributes are private and both are String objects.

* Modify HtmlWelcome servlet to add a link to iCarnegie in the welcome message. Note that this will need the following:
  + Create an instance of the HtmlAnchor class. The constructor will need the display text and the target location to be specified. For a link to iCarnegie in the welcome message, the display text is the string "iCarnegie" and the target location is the string "www.icarnegie.com".
  + Send the buildHtml message to the HtmlAnchor object created above. Note that after execution, the buildHtml message returns the HTML code for the hyperlink in the form of a String object.
  + Construct the welcome message with the hyperlink. Note that you will need to use string concatenation to build the welcome message from the following parts:
    - The string "Welcome to "
    - The string for the hyperlink returned by the buildHtml message to the HtmlAnchor object as above
    - The string "!"
  + Send the addText message to the HtmlPage object as before to add the welcome message to the body of the Web page.
* Note that to compile the HtmlWelcome servlet that uses the HtmlPage *and* HtmlAnchor classes, the source files for the servlet and *both* the HtmlPage and HtmlAnchor classes must be specified when you use the *javac* command for compilation. Further, to run this version of HtmlWelcome servlet, the class files for the servlet and *both* the HtmlPage and HtmlAnchor classes must be loaded into the iCarnegie Servlet Workbench. You may use the same form that was used in [2.1.4 Elements of a Java Servlet](javascript:ContentByName('pg-servlet-elements');) to invoke the servlet.
* The files *[HtmlAnchor.java](http://www.icarnegie.com/content/SSD/SSD1/2.1/normal/pg-intro-java-oop/pg-oop-fund/pg-htmlpage-class/HtmlAnchor.java" \t "externalWindow)* and *[HtmlWelcomeLink.java](http://www.icarnegie.com/content/SSD/SSD1/2.1/normal/pg-intro-java-oop/pg-oop-fund/pg-htmlpage-class/HtmlWelcomeLink.java" \t "externalWindow)* are provided as sample solution to this design task. Note that the sample solution names the modified HtmlWelcome servlet as HtmlWelcomeLink.

**2.2.5 Using Class Documentation**

When working with object-oriented programs in Java, you may design classes that others must use or you may need to use classes designed by others. Good programming practices dictate that one must not need to examine the source code for a class to be able to use it in a program. You saw in [2.1.6 Guidelines for Java Development](javascript:ContentByName('pg-guidelines-java');), when writing the Welcome servlet, you can refer to the [Java Servlet Specification](http://java.sun.com/products/servlet/2.3/javadoc/index.html" \t "external) site to explore the HttpServlet class in detail. You can generate similar documentation for classes you write using the Javadoc tool we read about in [2.1.6 Guidelines for Java Development](javascript:ContentByName('pg-guidelines-java');). Similarly, you may be expected to refer to such documentation when you use classes written by others.

In this section, we will learn how to use a class in a program by referring to the documentation for the class. We will modify the TravelRequest servlet from [2.1.5 Planning Servlet Development](javascript:ContentByName('pg-servlet-plan');) to use an object of the HtmlImage class. We provide the *.class* file and the documentation for the HtmlImage class.

**The HtmlImage Class**

Our goal is to re-write the TravelRequest servlet from [2.1.5 Planning Servlet Development](javascript:ContentByName('pg-servlet-plan');) as seen in Listing 1. We are told that we can separate all HTML code generation from the servlet if we use both HtmlPage and HtmlImage classes.

|  |  |
| --- | --- |
| 1:  2:  3:  4:  5:  6:  7:  8:  9: 10: 11: 12: 13: 14: 15: 16: 17: 18: 19: 20: 21: 22: 23: 24: 25: 26: 27: 28: 29: 30: 31: 32: 33: 34: 35: 36: 37: 38: 39: 40: 41: 42: 43: 44: 45: 46: 47: 48: 49: 50: 51: 52: 53: 54: 55: 56: 57: 58: 59: 60: 61: 62: 63: 64: 65: 66: 67: 68: 69: 70: 71: 72: | import java.io.\*;  import javax.servlet.\*;  import javax.servlet.http.\*;  public class TravelRequest extends HttpServlet {  public void doPost(  HttpServletRequest request,  HttpServletResponse response)  throws ServletException, IOException {  /\*\*  \* Indicate the content type (for example, text/html),  \* being returned by the response  \*/  response.setContentType("text/html");  /\*\*  \* Retrieve an output stream to use to send data  \* to the client  \*/  PrintWriter out = response.getWriter();  /\*\*  \* Get the user input from the form :  \* name and destination  \*/  String destination = request.getParameter("Destination");  String name = request.getParameter("Name");  /\*\*  \* Start by building the web page header  \*/  out.println(  "<!DOCTYPE HTML PUBLIC '-//W3C//DTD HTML 4.01 Transitional//EN'>");  out.println("<HTML>");  out.println("<HEAD>");  out.println("<TITLE>iCarnegie Travel Agency</TITLE>");  out.println("</HEAD>");  out.println("<BODY>");  /\*\*  \* Add the image  \*/  out.println("<IMG src='/" + destination + "' alt='"  + destination + "'>");  /\*\*  \* Add the confirmation message, with the name  \*/  out.println("Hello, " + name + "!");  out.println(  "Thanks for your request. You will be contacted shortly.");  /\*\*  \* End by building the web page footer  \*/  out.println("</BODY>");  out.println("</HTML>");  } // end-doPost  } |
| **Listing 1** *TravelRequest.java* | |

As our first step, we re-write the source code for the TravelRequest servlet to use the HtmlPage class. Listing 2 shows the modified source for the TravelRequest servlet that uses the HtmlPage class to generate the response Web page. You can follow the HtmlWelcome servlet example from [2.2.5 The HtmlPage Class](javascript:ContentByName('pg-htmlpage-class');) to understand the changes to Listing 1.

|  |  |
| --- | --- |
| 1:  2:  3:  4:  5:  6:  7:  8:  9: 10: 11: 12: 13: 14: 15: 16: 17: 18: 19: 20: 21: 22: 23: 24: 25: 26: 27: 28: 29: 30: 31: 32: 33: 34: 35: 36: 37: 38: 39: 40: 41: 42: 43: 44: 45: 46: 47: 48: 49: 50: 51: 52: 53: 54: 55: 56: 57: 58: 59: 60: 61: 62: 63: | import java.io.\*;  import javax.servlet.\*;  import javax.servlet.http.\*;  public class TravelRequest extends HttpServlet {  public void doPost(HttpServletRequest request,  HttpServletResponse response)  throws ServletException, IOException {  /\*\*  \* Indicate the content type (for example, text/html),  \* being returned by the response  \*/  response.setContentType("text/html");  /\*\*  \* Retrieve an output stream to use to send data  \* to the client  \*/  PrintWriter out = response.getWriter();  /\*\*  \* Get the user input from the form :  \* name and destination  \*/  String name = request.getParameter("Name");  String destination = request.getParameter("Destination");  /\*\*  \* Create an instance of HtmlPage class  \*/  HtmlPage myPage = new HtmlPage();  /\*\*  \* Set the title for the response Web page  \*/  myPage.setTitle("iCarnegie Travel Agency");  /\*\*  \* Add the image  \*/  myPage.addText(  "<IMG src='/" + destination + "' alt='"  + destination + "'>");  /\*\*  \* Add the confirmation message with the name  \*/  myPage.addText("Hello, " + name + "!");  myPage.addText(  "Thanks for your request. You will be contacted shortly.");  /\*\*  \* Write the response  \*/  out.println(myPage.buildHtml());  } // end-doPost  } |
| **Listing 2** *TravelRequest.java* | |

**Documentation for the HtmlImage Class**

As seen from the following line of Listing 2, we are still one more step away from separating all HTML code from the servlet.

myPage.addText( "<IMG src='/" + destination + "' alt='" + destination + "'>");

Let us now examine the [documentation](http://www.icarnegie.com/content/SSD/SSD1/2.1/normal/pg-intro-java-oop/pg-oop-fund/pg-use-class/HtmlImage/index.html" \t "externalWindow) for HtmlImage class.

If you browse through all parts of the documentation, you will find many terms that you do not understand. Note that as you learn more Java and object-oriented programming, you will understand more information presented there. At this time, we will only focus on information related to concepts covered so far in Units 1 and 2.

The section titled *Constructor Summary* indicates that the class has only one constructor, which takes two String parameters. Looking further in the section titled *Constructor Detail*, we learn that:

* The first parameter to the constructor is a String that represents the name of the image file to be used for the **src** attribute of the **IMG** tag.
* The second parameter to the constructor is a String that represents the alternate text to be used for the **alt** attribute of the **IMG** tag.

The section titled *Method Summary* indicates that the class has only one method buildHtml, which takes no parameters. Looking further in the section titled *Method Detail*, we learn that buildHtml is a public method that returns a String representing the HTML code for the **IMG** tag with the **src** and **alt** attributes.

On the basis of above information, we can conclude that:

* The HtmlImage class is a simple class with one constructor and one method
* The values for the **src** and **alt** attributes are passed as parameters to the constructor when the HtmlImage class is instantiated
* We can send the buildHtml message to the HtmlImage class to obtain the HTML code for the **IMG** tag that we can then add to our Web page.

**Using the HtmlImage Class**

Listing 3 shows the source for the TravelRequest servlet modified further to use the HtmlImage class.

|  |  |
| --- | --- |
| 1:  2:  3:  4:  5:  6:  7:  8:  9: 10: 11: 12: 13: 14: 15: 16: 17: 18: 19: 20: 21: 22: 23: 24: 25: 26: 27: 28: 29: 30: 31: 32: 33: 34: 35: 36: 37: 38: 39: 40: 41: 42: 43: 44: 45: 46: 47: 48: 49: 50: 51: 52: 53: 54: 55: 56: 57: 58: 59: 60: 61: 62: 63: 64: 65: 66: 67: 68: 69: | import java.io.\*;  import javax.servlet.\*;  import javax.servlet.http.\*;  public class TravelRequest extends HttpServlet {  public void doPost(HttpServletRequest request,  HttpServletResponse response)  throws ServletException, IOException {  /\*\*  \* Indicate the content type (for example, text/html),  \* being returned by the response  \*/  response.setContentType("text/html");  /\*\*  \* Retrieve an output stream to use to send data  \* to the client  \*/  PrintWriter out = response.getWriter();  /\*\*  \* Get the user input from the form :  \* name and destination  \*/  String name = request.getParameter("Name");  String destination = request.getParameter("Destination");  /\*\*  \* Create an instance of HtmlPage class  \*/  HtmlPage myPage = new HtmlPage();  /\*\*  \* Set the title for the response Web page  \*/  myPage.setTitle("iCarnegie Travel Agency");  /\*\*  \* Create an instance of the HtmlImage class  \*/  HtmlImage destinationImage =  new HtmlImage( "/" + destination, destination );  /\*\*  \* Add the image to the Web page  \*/  myPage.addText( destinationImage.buildHtml() );  /\*\*  \* Add the confirmation message with the name  \*/  myPage.addText("Hello, " + name + "!");  myPage.addText(  "Thanks for your request. You will be contacted shortly.");  /\*\*  \* Write the response  \*/  out.println( myPage.buildHtml() );  } // end-doPost  } |
| **Listing 3** *TravelRequest.java* | |

First, we create an instance of the HtmlImage class as seen from the following line of Listing 3:

HtmlImage destinationImage = new HtmlImage( "/" + destination, destination);

Note that we pass the same parameter for both the name of the name of the image file and the alternate text. Recall from [2.1.5 Planning Servlet Development](javascript:ContentByName('pg-servlet-plan');) that we did not have any specification for the alternate text.

Next, we add the image to the Web page as seen from the following line of Listing 3:

myPage.addText(destinationImage.buildHtml());

This line of code really involves two steps:

* Send the buildHtml message to destinationImage, which is an instance of the HtmlImage class obtained in the previous step.
* Send the addText message to myPage, which is an instance of the HtmlPage class we are using to construct our Web page. The parameter passed to the addText message is the String returned when the buildHtml message sent to destinationImage above completes execution.

The source code for TravelRequest servlet must now be prepared for execution with the iCarnegie Servlet Workbench to verify the changes to the source code. This involves the following steps:

* Download *HtmlImage.class* file into the same directory where you are working.
* Compile *TravelRequest.java* with the *javac* command. Recall from [2.2.5 The HtmlPage Class](javascript:ContentByName('pg-htmlpage-class');) that you must specify *HtmlPage.java* on the same command line if you did not compile the HtmlPage class before.
* Add the path to the directory where you are working to the system's **classpath**. This is so that the compiler can locate the *.class* file for the HtmlImage class. Note that this will also allow the compiler to locate the *.class* file for the HtmlPage class in case you already had compiled it before and chose not to specify the file *HtmlPage.java* on the *javac* command line for compiling *TravelRequest.java*.
* Load the files *TravelRequest.class and HtmlImage.class* into the workbench with theLoad Javaoption from the Actionsmenu of the workbench. You will also need to load the file *HtmlPage.class* if not already loaded.
* Select the file *TravelRequstForm.html* from the Content directory within the workbench, and then click Open in Browseron the Actionsmenu.

Here is something you can try:

Please refer to the [Appendix D. HTML Classes](javascript:ContentByName('pg-html-classes');). Follow the link for class documentation and try to understand the class definition for the HtmlTable class. You will find it useful when working on assessments later in the course.

You have seen in this section that even though you did not have the *.java* file for the HtmlImage class, you were able to refer to the class documentation and understand the function of the class. You were able to create an instance of the class and send the appropriate message to that instance to achieve the desired change in your program. Thus, you did not need to examine the source code for the class to be able to use it.

**2.2.6 The AlgaeColony Class**

In [2.2.1 Designing Classes](javascript:ContentByName('pg-design-class')), we worked through a process for designing classes, which involved writing the specification for the class in English. Further, in [2.2.2 Transforming English Specification into Java](javascript:ContentByName('pg-transform-class')), we saw how to map the English Specification for the class to Java source code. Applying the concepts from these sections and the readings from the textbook, we will attempt the preliminary design of AlgaeColony class used by yet another version of the simulation program.

To proceed with the design, download the [new version](http://www.icarnegie.com/content/SSD/SSD1/2.1/normal/pg-intro-java-oop/pg-oop-fund/pg-algae-class/Alife-no-inheritance.zip) of the simulation program. Load the *.class* and content files into the workbench and open the form in *initialAlgaeFishCrocodile.html*. You may recall the initial conditions you chose to run the simulation program the very first time in [2.1.1 Programming with Objects](javascript:ContentByName('pg-objects-java')). You are running a different version of the simulation program at this time, but most of the behavior you will observe is similar. Try some of the conditions you chose earlier and observe the behavior of the AlgaeColony as best as you can to prepare for designing the class that models the algae colonies in the simulation.

For now, please make a note that the design of the simulation program requires all organisms in the simulation, such as AlgaeColony and Catfish, to exhibit the behavior of the class LivingBeing. If you refer to the [documentation](http://www.icarnegie.com/content/SSD/SSD1/2.1/normal/pg-intro-java-oop/pg-oop-fund/pg-algae-class/LivingBeing/index.html" \t "externalWindow) for the class LivingBeing, you will notice that it is an abstract class. This requires that the AlgaeColony class MUST define certain behaviors defined by the LivingBeing class.

**Designing the AlgaeColony Class**

To design the class that models AlgaeColony, we will follow the same steps as in [2.2.1 Designing Classes](javascript:ContentByName('pg-design-class')).

First, we give some thought to the behaviors of object instances.

We are given the following information about the simulation program that relates to the behavior of AlgaeColony:

* The simulation program generates random amounts of sunlight per cell, a portion of which is converted to life energy by the AlgaeColony.
* The energy of any organism at any given time must be between the minimum amount needed to stay alive and the maximum amount of energy that it can carry. If the energy carried by the organism falls below the minimum, it will die.
* An AlgaeColony that is dead disappears from the lake.
* An AlgaeColony that is alive changes color based on the level of energy it carries.
* When an AlgaeColony is eaten by a Catfish, it must give up some energy.
* All organisms expend some energy to live.
* Every organism belongs to a certain species of living beings.
* Every organism has a certain age measured in terms of blocks of time. One block of time is equivalent to one step of simulation.

Further, we can observe the following about the behavior of AlgaeColony from running the simulation:

* An AlgaeColony does not move.
* The color of an AlgaeColony changes between three shades of green and black.
* When Catfish are located close to an AlgaeColony, the Catfish move differently.

We can thus identify the following behaviors at this time:

* *Constructor*. Create an instance of the AlgaeColony class.
* *Live*. Live a block of time or a step of simulation.
* *Die*. Update living status. Give up energy. If eaten by Catfish, relinquish energy.

Next, we identify the attributes of object instances:

* *Location*. Location in the grid that represents the lake, expressed in terms of the row and column
* *Life status*. Whether it is dead or alive
* *Energy level*. The amount of energy it has
* *Color*. Can be several shades of green based on the level of energy if it is alive or black if it is dead
* *Identification*. Some form of identification for the object instance, which could be a name

Finally, we identify class attributes:

* Species
* Minimum energy required by newborn AlgaeColony to stay alive
* Maximum energy that can be carried by a newborn AlgaeColony
* Amount of energy used to live a block of time

Before we proceed with the next step in the design process, we need to give some thought to how AlgaeColony interact with the other objects in the simulation program. Some of these objects are visible such as the Catfish and the Crocodile. There are some objects that are not visible. One such invisible object must be the servlet invoked when we submit the form in *initialAlgaeFishCrocodile.html* to start the simulation. We can guess that there may be others from the following observations:

* The way different organisms are represented during their lifetime varies. For instance, Catfish and Crocodile are represented by images that disappear when they are dead. The color of their images does not change during their lifetime. On the other hand, AlgaeColony is represented by a change of color.
* An AlgaeColony must obtain sunlight.
* AlgaeColony objects must be created in the location(s) specified by the user.

We are told that the simulation program uses an instance of the Simulation class to keep track of the different steps of the simulation and hold the various organisms. We are also given the [documentation](http://www.icarnegie.com/content/SSD/SSD1/2.1/normal/pg-intro-java-oop/pg-oop-fund/pg-algae-class/Simulation/index.html" \t "externalWindow) for this class, which shows that an AlgaeColony object can send the getSunlightmessage to the Simulation object to obtain sunlight at a specific location. The documentation further shows that the Simulation class defines the attributes COLOR and IMAGE that represent the mechanism used to display an organism.

**English Specification for AlgaeColony Class**

After giving some thought to the design, we can summarize the English specification for the AlgaeColony class as follows:

Behaviors required since class AlgaeColony must implement the behaviors of class LivingBeing:

* *liveALittle*. Live a block of time. Get sunlight for its location from the Simulation object and convert part of it to energy. Relinquish necessary energy to live and die if the energy falls below the minimum required to stay alive. Returns nothing.
* *getRow*. Return the row in which AlgaeColony is located as an integer.
* *getColumn*. Return the column in which AlgaeColony is located as an integer.
* *getAge*. Return the age of AlgaeColony as an integer.
* *getName*. Return the unique name identifying the AlgaeColony as a String.
* *getImage*. Return an image representing the AlgaeColony.
* *getColor*. Return the color of the AlgaeColony as a String.
* *getEnergy*. Return the current energy level as an integer.
* *isDead*. Return the living status of the AlgaeColony as a boolean.
* *getDisplayMechanism*. Return the mechanism used for displaying the AlgaeColony, which is either the name of an image file or a constant defined in the Simulation class.
* *getSpecies*. Return the species of the AlgaeColony as a String.

Behaviors specific to AlgaeColony:

* *Constructor*. Create an instance of AlgaeColony at the specified row and column of the grid. It will be have a certain amount of energy and an age 0. It will belong to the "Algae" species. It can be identified by a unique name. The maximum energy it can carry must be set to the maximum energy allowed for a newborn AlgaeColony. Similarly, the minimum energy it needs to stay alive must be set to the minimum energy required for an AlgaeColony to stay alive.
* *die*. Update living status to being dead. Returns nothing.
* *giveUpEnergy*. Relinquish specified amount of energy if possible. Returns the energy that can be given up as an integer.
* *getMinEnergy*. Return the minimum energy needed to stay alive as an integer.
* *getMaxEnergy*. Return the maximum energy that can be carried as an integer.
* *setEnergy*. Set the energy level as specified by the integer value passed as a parameter.

We can see that the behaviors getMinEnergyand getMaxEnergyallow the behavior setEnergyto verify that the specified energy level is within the permissible range. Also, the behavior dieallows setEnergyto update the living status in case the energy level is below the threshold to stay alive. The behavior giveUpEnergy depletes the energy level when AlgaeColony is eaten by Catfish.

Constant class attributes:

* *ALIVE*. This private attribute represents the status of AlgaeColony that is alive as a String.
* *DEAD*. This private attribute represents the status of AlgaeColony that is dead ss a String.
* *SPECIES*. This private attribute represents the species of AlgaeColony as a String.
* *BABY\_MAX\_ENERGY*. This private attribute represents the maximum energy that can be carried as an integer.
* *BABY\_MIN\_ENERGY*. This private attribute represents the minimum energy needed to stay alive as an integer.
* *ENERGY\_TO\_LIVE*. This private attribute represents the energy required to live one block of time as an integer.

Constant instance attributes:

* *name*. This private attribute represents a unique name as a String.

Instance attributes:

* *row*. This private attribute represents the row identifying the location as an integer.
* *column*. This private attribute represents the column identifying the location as an integer.
* *deadOrAlive*. This private attribute represents the living status as a String.
* *energy*. This private attribute represents the current energy level as a String.
* *age*. This private attribute represents the age as an integer.
* *simulation*. This private attribute represents the Simulation to which the AlgaeColony belongs.
* *minEnergy*. This private attribute represents the minimum energy needed to survive.
* *maxEnergy*. This private attribute represents the maximum energy that can be carried.

Class attributes:

* *nAlgaeCreated*. This private attribute represents the number of algae created so far as an integer. This count helps the Simulation class keep track of the total number of AlgaeColony objects created in the simulation.

**Mapping English Specification to Java Code**

As seen in [2.2.2 Transforming English Specification into Java](javascript:ContentByName('pg-transform-class')), we arrive at the following class definition for AlgaeColony when we map our English specification to Java code. Note that this class definition is not complete and defers some of actual coding until later.

|  |  |
| --- | --- |
| 1:   2:   3:   4:   5:   6:   7:   8:   9:   10:  11:  12:  13:  14:  15:  16:  17:  18:  19:  20:  21:  22:  23:  24:  25:  26:  27:  28:  29:  30:  31:  32:  33:  34:  35:  36:  37:  38:  39:  40:  41:  42:  43:  44:  45:  46:  47:  48:  49:  50:  51:  52:  53:  54:  55:  56:  57:  58:  59:  60:  61:  62:  63:  64:  65:  66:  67:  68:  69:  70:  71:  72:  73:  74:  75:  76:  77:  78:  79:  80:  81:  82:  83:  84:  85:  86:  87:  88:  89:  90:  91:  92:  93:  94:  95:  96:  97:  98:  99:  100: 101: 102: 103: 104: 105: 106: 107: 108: 109: 110: 111: 112: 113: 114: 115: 116: 117: 118: 119: 120: 121: 122: 123: 124: 125: 126: 127: 128: 129: 130: 131: 132: 133: 134: 135: 136: 137: 138: 139: 140: 141: 142: 143: 144: 145: 146: 147: 148: 149: 150: 151: 152: 153: 154: 155: 156: 157: 158: 159: 160: 161: 162: 163: 164: 165: 166: 167: 168: 169: 170: 171: 172: 173: 174: 175: 176: 177: 178: 179: 180: 181: 182: 183: 184: 185: 186: 187: 188: 189: 190: 191: 192: 193: 194: 195: 196: 197: 198: 199: 200: 201: 202: 203: 204: 205: 206: 207: 208: 209: 210: 211: 212: 213: 214: 215: 216: 217: 218: 219: 220: 221: 222: 223: 224: 225: 226: 227: 228: 229: 230: 231: 232: 233: 234: 235: 236: 237: 238: 239: 240: 241: 242: 243: 244: 245: 246: 247: 248: 249: 250: 251: 252: 253: 254: 255: 256: 257: 258: 259: 260: | /\*\*  \* AlgaeColony does not move. If there is sunlight, a portion of the solar  \* energy will be converted into life-energy.  \*  \* @author iCarnegie av  \*/  public class AlgaeColony extends LivingBeing {  /\*\*  \* The algae is born "alive".  \* Then it dies, becoming a corpse.  \*/  private static final String ALIVE = "alive";  /\*\*  \* The algae is born "alive".  \* Then it dies, becoming a "dead" corpse.  \*/  private static final String DEAD = "dead";  /\*\*  \* Lowest possible energy needed for a baby to survive.  \*/  private static final int BABY\_MIN\_ENERGY = 5;  /\*\*  \* Maximum energy that a baby can store.  \*/  private static final int BABY\_MAX\_ENERGY = 255;  /\*\*  \* Amount of energy needed to live for a block of time.  \*/  private static final int ENERGY\_TO\_LIVE = 1;  /\*\*  \* Name of species  \*/  private static final String SPECIES = "Algae";  /\*\*  \* Row-wise location of the algae  \*/  private int row;  /\*\*  \* Column-wise location of the algae  \*/  private int column;  /\*\*  \* Is the algae dead or alive?  \*/  private String deadOrAlive;  /\*\*  \* Amount of energy the algae has.  \*/  private int energy;  /\*\*  \* Age expressed as blocks of time lived  \*/  private int age = 0;  /\*\*  \* Name of this algae.  \*/  private final String name;  /\*\*  \* The simulation to which this algae belongs.  \* This is needed so the algae can send a message  \* to simulation and ask  \* for prey (or predator) in the neighboring locations.  \* Prey is food. Food is good!  \*/  private Simulation simulation;  /\*\*  \* Minimum energy level needed to survive.  \* The minimum could increase as the individual grows.  \*/  private int minEnergy;  /\*\*  \* Maximum energy level that the algae could carry.  \* The maximum could change as the individual grows.  \*/  private int maxEnergy;  /\*\*  \* Number of Algae objects created so far.  \*/  private static int nAlgaeCreated = 0;  /\*\*  \* Constructor. Initialize an algae to start life at a specified  \* location with a specified energy. If location is out of bounds,  \* locate the algae at the nearest edge.  \*  \* @param initialRow - the row at which the algae is located  \* @param initialColumn - the column at which the algae is located  \* @param initialSimulation - the simulation that the algae belongs to  \*/  public AlgaeColony(  int initialRow,  int initialColumn,  Simulation initialSimulation) {  name = "dummyName";  }  /\*\*  \* Get the row at which the algae is located  \*  \* @return - the row of the algae's location.  \*/  public int getRow() {  return 1;  }  /\*\*  \* Get the column at which the algae is located  \*  \* @return - the column of the algae's location.  \*/  public int getColumn() {  return 1;  }  /\*\*  \* Get the algae's age  \*  \* @return the age of the algae expressed in blocks of time  \*/  public int getAge() {  return 0;  }  /\*\*  \* Get the name of this algae  \*  \* @return the name of the algae.  \*/  public String getName() {  return "dummyName";  }  /\*\*  \* Get the filename that contains the image of the algae  \*  \* @return the image of the algae.  \*/  public String getImage() {  return "/blank.jpg";  }  /\*\*  \* Get the minmum energy needed to live.  \*  \* @return the minimum energy needed for the algae to live.  \*/  private int getMinEnergy() {  return BABY\_MIN\_ENERGY;  }  /\*\*  \* get the maximum energy that the algae can carry.  \*  \* @return the maximum energy the algae can carry.  \*/  private int getMaxEnergy() {  return BABY\_MAX\_ENERGY;  }  /\*\*  \* Get the energy currently carried by the algae.  \*  \* @return current energy level of the organism  \*/  public int getEnergy() {  return 100;  }  /\*\*  \* Sets energy level.  \* If new energy level is less than minimum energy level, the organism dies.  \* New energy level is capped at maximum energy level.  \*/  private void setEnergy(int newEnergy) {  }  /\*\*  \* Die: Change the deadOrAlive to DEAD.  \*/  private void die() {  }  /\*\*  \* Is the algae dead?  \*  \* @return <code>true</code> if dead. <code>false</code>, otherwise.  \*/  public boolean isDead() {  return false;  }  /\*\*  \* The mechanism to display Algae is to use its color.  \*  \* @return a constant defined in {@link Simulation#COLOR Simulation} class  \*/  public String getDisplayMechanism() {  return Simulation.COLOR;  }  /\*\*  \* Get the species that the algae belongs to  \*  \* @return a string indicating the species.  \*/  public String getSpecies() {  return SPECIES;  }  /\*\*  \* Get the color of Algae  \*  \* @return - the color as a string in hexademinal notation  \*/  public String getColor() {  return "green";  }  /\*\*  \* Algae is being eaten up.  \* So, relinquish energy up to the amount requested.  \* If no energy remains, die.  \*  \* @param energyWanted - amount of energy requested - expressed as int.  \* @return - the amount of energy that algae can give up.  \* If the requested energy is greater than the available energy,  \* only the available energy will be given up.  \*/  public int giveUpEnergy(int energyWanted) {  return 100;  }  /\*\*  \* Algae lives its life. May gain or lose energy.  \*/  public void liveALittle() {  }  } |
| Listing 1 *AlgaeColony.java* | |

Try to follow the source code in Listing 1 to understand how it maps to the English specification. Note that as seen in [2.2.2 Transforming English Specification into Java](javascript:ContentByName('pg-transform-class')), we have forced some of the method definitions to return a value as specified by the return type of the method. This allowed us to verify that the class definition compiles successfully.

Note also that the line of code that starts the class definition includes extends LivingBeing. Recall that the design of the simulation program requires that all organisms in the simulation exhibit the behavior of the class LivingBeing. Including extends LivingBeing in the class definition for AlgaeColony achieves this.

Here is a class design task you can try:

Try to design the class that models the Catfish in the simulation. Here are a few observations that can help you start the design process:

A Catfish is an organism that shares some attributes and behaviors with the AlgaeColony. For instance, a Catfish shares the following constant class attributes with the AlgaeColony

* *ALIVE*. Status of Catfish that is alive represented as a String
* *DEAD*. Status of Catfish that is dead represented as a String
* *SPECIES*. Species of Catfish represented as a String
* *BABY\_MAX\_ENERGY*. Maximum energy that can be carried represented as an integer
* *BABY\_MIN\_ENERGY*. Minimum energy needed to stay alive represented as an integer

Constant instance attributes:

* *name*. Unique name represented as a String

Instance attributes:

* *row*. Row identifying the location represented as an integer
* *column*. Column identifying the location represented as an integer
* *deadOrAlive*. Living status represented as a String
* *energy*. Current energy level represented as a String
* *age*. Age represented as an integer
* *simulation*. Simulation to which the Catfish belongs
* *minEnergy*. Minimum energy needed to survive
* *maxEnergy*. Maximum energy that can be carried

Class attributes:

* *nCatfishCreated*. Number of Catfish created so far represented as an integer, to help the Simulation object keep track of the total number of Catfish in the simulation

Similarly, it is easy to see that the Catfish would also exhibit the following behaviors:

* *Constructor*. Create an instance of the Catfish class.
* *liveALittle*. Live a block of time.
* *die*. Update living status to being dead. Returns nothing.
* *getRow*. Return the row in which Catfish is located as an integer.
* *getColumn*. Return the column in which Catfish is located as an integer.
* *getAge*. Return the age of Catfish as an integer.
* *getName*. Return the unique name identifying the Catfish as a String.
* *getImage*. Return an image representing the Catfish.
* *getMinEnergy*. Return the minimum energy needed to stay alive as an integer.
* *getMaxEnergy*. Return the maximum energy that can be carried as an integer.
* *getEnergy*. Return the current energy level as an integer.
* *isDead*. Return the living status of the Catfish as a boolean.
* *getDisplayMechanism*. Return the mechanism used for displaying the Catfish as a String.
* *getSpecies*. Return the species of the Catfish as a String.
* *getColor*. Return the color of the Catfish as a String.
* *setEnergy*. Set the energy level as specified by the integer value passed as a parameter.

So, your first step should be to think about attributes and behavior common to AlgaeColony and Catfish. Your next step should be to think about attributes and behavior specific to Catfish. For instance, Catfish can move right, left, up or down in the grid. This suggests the following about the Catfish class:

Instance attribute:

* *direction*. The direction in which the Catfish is facing represented as a String

It also suggests the following for the Catfish class:

Constant class attributes:

* *RIGHT*. Direction of Catfish facing right represented as a String
* *LEFT*. Direction of Catfish facing left represented as a String
* *UP*. Direction of Catfish facing up represented as a String
* *DOWN*. Direction of Catfish facing down represented as a String

We know that Catfish can move and are capable of eating AlgaeColony. This suggests the following behaviors specific to Catfish:

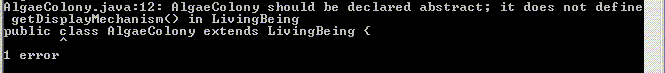
* *moveToRow*. Move the Catfish to the row specified by the integer value passed as a parameter. Returns the row where the Catfish is moved as an integer.
* *moveToColumn*. Move the Catfish to the column specified by the integer value passed as a parameter. Returns the column where the Catfish is moved as an integer.
* *isHungry*. Check if the current energy level of the Catfish is less than twice the amount needed for survival. Returns a boolean.
* *getDirection*. Check the direction in which the Catfish is facing. Returns a String.

Clearly, there are other behaviors that need consideration. For instance, the Catfish need to locate AlgaeColony and eat them. This suggests behaviors that model Catfish looking for food, trying to eat food if available or simply swimming to a new location. Thus, a Catfish looking for food would actually need to look for neighboring AlgaeColony in the lake. This suggests the following behavior:

* *lookForFoodInNeighborhood*. Try to locate any AlgaeColony close by. Relinquish some energy in the process. Note that this behavior requires a way to store the neighboring AlgaeColony objects. Your knowledge of Vectors from [2.2.3 Lifecycle of Objects](javascript:ContentByName('pg-objects-lifecycle');) can help you here.

Approach this design task by following the same steps as done for AlgaeColony:

* Run the simulation to observe the behavior of the Catfish as best as you can.
* Give some thought to the behaviors and attributes to be defined by the Catfish class.
* Write down the English specification for the design.
* Map the English specification to Java code. Verify that the class definition compiles successfully. Note that you will need to add the path of the directory where you are working to the system's **classpath** so that the compiler can locate the necessary *.class* files.
* Remember that the Catfish class MUST define certain behaviors that are defined by the LivingBeing class. If this is not done, you will see compile-time errors such as the following if getDisplayMechanismis not defined by the AlgaeColony class:



[Catfish.java](http://www.icarnegie.com/content/SSD/SSD1/2.1/normal/pg-intro-java-oop/pg-oop-fund/pg-algae-class/Catfish.java) provides a partial implementation of Catfish class as discussed above.

**2.3 Fundamentals of Java**

In the previous module, [2.2 Fundamentals of Object-Oriented Programming](javascript:ContentByName('pg-oop-fund');), we designed the AlgaeColony class and parts of the Catfish class. In the exercise, you were asked to complete the design of Catfish class and Crocodile class. We have not yet implemented any of the methods. We will complete the method implementations in this module.

Since the Java implementation requires a number of concepts that are new to you, we will follow the pattern of introducing new concepts and using the AlgaeColony implementation as an example to illustrate the concepts.

We also designed the BankAccount and AccountsLedger classes for the bank teller example in [2.2 Fundamentals of Object-Oriented Programming](javascript:ContentByName('pg-oop-fund');). At this time, please download the complete [BankAccount.java](http://www.icarnegie.com/content/SSD/SSD1/2.1/normal/pg-intro-java-oop/pg-oop-fund/pg-design-class/BankAccount/BankAccount.java) and [AccountsLedger.java](http://www.icarnegie.com/content/SSD/SSD1/2.1/normal/pg-intro-java-oop/pg-oop-fund/pg-design-class/BankAccount/AccountsLedger.java) and inspect them for relevant examples as you read this module.

**2.3.1 Data Types and Variables**

Information plays a central role in computer programming. Indeed, it is safe to say that the focus of all programs is the manipulation of information. Given the central role that information plays, it is important that you understand the various mechanisms that Java provides for the classification, storage and access of information. In this section of the course, you will learn how Java categorizes information and in what contexts such a categorization is important. You will also learn about Java's rules and process for storing information and making use of that stored information at a later point.

**2.3.1.1 Data Types**

* [Data Types](http://www.icarnegie.com/content/SSD/SSD1/2.1/normal/pg-intro-java-oop/pg-java-fund/pg-data-types-variables/pg-data-types/pg-data-types.html" \l "datatypes#datatypes)
* [Primitive Types](http://www.icarnegie.com/content/SSD/SSD1/2.1/normal/pg-intro-java-oop/pg-java-fund/pg-data-types-variables/pg-data-types/pg-data-types.html" \l "prim#prim)
* [Reference Types](http://www.icarnegie.com/content/SSD/SSD1/2.1/normal/pg-intro-java-oop/pg-java-fund/pg-data-types-variables/pg-data-types/pg-data-types.html" \l "reference#reference)

**Data Types**

In programming parlance, the terms *type* and *data type* refer to a specific kind of information. This may seem strange initially, but if you consider this concept in the context of how OOP classifies real world information, it should become easier to grasp. Recall that objects (both in Java and in the real world) have attributes and methods. Attributes are characteristics of objects and methods are the operations that objects can perform. Each attribute may take on specific kinds of information. In programming terms, we would say that an attribute has a specific data type.

Consider the example of a Chair class. Attributes of Chair could include its height, its color, its maximum load, and the presence of arms. In instantiating a Chair object, we might consider these attributes in the form of questions:

* What is the chair's height?
* What color is the chair?
* What is the maximum load of the chair?
* Does the chair have arms?

Each of these questions has a limited set of acceptable answers. Unacceptable answers can be either ambiguous or incorrect. One meter could certainly be the answer to "What is the chair's height?" but would not be an acceptable answer to "What color is the chair?" Similarly, "blue" can describe the chair's color, but not its height. Each Chair object's attribute must be of a certain kind of information; that is, each attribute has a specific data type. Consider the following table:

|  |  |
| --- | --- |
| **Attribute** | **Data Type** |
| height | linear measurement |
| chair color | color |
| maximum load | weight |
| arms | yes/no |

**Table 1** *Each attribute has a data type.*

In the table, we have assigned data types to attributes of Chair. These data types are probably acceptable for a casual consideration of the Chair class, or even for your own personal interactions with Chair objects in the real world. For more serious work with chair, however, these data types might cause problems. Should linear measurement be in metric or U.S. units? Must we say that the chair color is "blue," or could we also use RGB notation and say that it is "#0000FF"? As you will see, Java's data types are much stricter in this regard.

Data types are not limited to attributes; they are also very important when defining and calling methods. Imagine that our Chair is able have its height adjusted (like an office chair) — we will call it the setHeight method. In order to call setHeight, you would need to supply some information; specifically, you will need to provide a height. As our earlier discussion of attributes illustrates, only certain types of information would be acceptable: a linear measurement would make sense. In contrast, a color or a yes/no would not. In order to call setHeight, you must supply information of a specific type.

You have already had some experience with data types, although you were probably not aware of it at the time. Think back to the attributes of HTML tags that you learned in [Unit 1. The World Wide Web](javascript:ContentByName('pg-www');). The bgcolor attribute of the BODY tag could only have a value of a certain type. If you supplied a value of an incorrect type, browsers would not display the page correctly. Similarly, the type attribute of the INPUT tag could be assigned a value of "text" or "radio," but not "multiple choice." This is true for all of the attributes that you learned—each has a limited set of values that will be interpreted correctly by a browser. In the case of HTML, these types are defined within the [HTML Specification](http://www.htmlhelp.com/reference/html40/" \t "external).

Java also has data types. Its use of types is much more fully developed — and much stricter — than HTML's. In fact, you have already encountered one case of Java's use of data types when you passed parameters to methods. In Java, the data type of a parameter is specified in the method declaration, and attempting to pass a parameter that is not of the specified data type will cause problems. Consider the method println. This method requires a String argument and supplying data of a different type causes a compilation error. If you care to test this, create a simple servlet with a println call and attempt to compile it. First, use the form:

println("This is a String");

This example should compile, because it passes a parameter of the proper data type to println. Now modify the code to use the following:

println(3);

As you might have anticipated, this second case will not compile. "This is a String" is an instance of an acceptable data type for println; the number 3 is not.

At this point, you should have a conceptual understanding of data types—there are specific kinds of information that we refer to as data types. In the context of attributes and methods, the data type specifies what kind of information can be used to assign or pass a value. Data types are an essential feature of the Java programming language.

We can broadly categorize Java's types into two kinds: *Primitive types* and *reference types*.

**Primitive Types**

Primitive types are the simplest kinds of information available in Java. These are the basic building blocks of Java and all other classes are in some way built on these fundamental types. One of the defining characteristics of a data type is its size in computer’s memory. Memory is measured in bits (***b****inary dig****it***). A binary digit can take two values, 0 or 1. A memory *bit* can store two states. Bits can be combined to store information that is more meaningful. Since certain types of information require a certain minimum number of bits, a type’s size is one of its important characteristics. Primitive data types are identified in Java by keywords — for example, the keyword int identifies items as integers, such as the number 3, while the keyword boolean identifies items as being either true or false. Java has eight primitive types; within this course, however, you will make use of only three primitive types. A full listing of Java's primitives can be found in your textbook; the three types that you will need to know are:

|  |  |  |  |
| --- | --- | --- | --- |
| **Type** | **Represents** | **Size** | **Example** |
| boolean | Boolean value (true or false) | 1 bit | true |
| double | Floating Point (from +-4.9E-324 to +-1.7976931348623157E+308) | 64 bits | 3.14 |
| int | Integer value (from -2,147,483,648 to 2,147,483,647) | 32 bits | 6 |

**Table 2** *Subset of primitive data types in Java*

As you might have guessed, if you had access to only these primitive building blocks, building a complete program would be very challenging indeed (although not impossible – many older languages required programmers to do just that). To avoid this burden, more advanced types are also available. You have already interacted with some of these kinds of data. The String class, for example, represents one of these types. Java refers to these advanced data types as *Reference Types*.

**Reference Types**

Every Java class is a reference type. Indeed, the term "reference type" derives from the fact that, to make use of this type of data, you "refer" to a class instance. Reference types range from very simple classes built solely from primitives to very complex types, such as HttpServlet, composed of other reference types. One relatively simple reference type is the String class. When working with text in Java, using individual characters is rarely appropriate. Instead, you will normally work with a "string" of characters, such as "My name is Thabo". Certain characteristics of a string of characters may be important to you, and there are certain things you may wish to do with this string of characters, such as replace certain characters or determine its length. In order to model a string of characters and allow one to interact with a string as a unit (rather than as a series of individual characters), the String class was created. As you might imagine, manipulating individual characters could quickly become tedious and confusing. By modeling a string of characters as a String class, working with strings of characters becomes more natural and less prone to error. These benefits represent the spirit of OOP.

All classes in the Java language can be used as reference types. This is true of classes provided in Java's packages and those that you create yourself. You have already worked with reference types in Java, although you were not aware of it. Review your submission for the [previous practical quiz](javascript:AssessmentByName('assm-qz-pr-oop-fund');) and see if you can find the following reference types:

* PrintWriter
* HttpServlet
* HttpServletResponse

In creating a class, you actually create a new type. In the [previous exercise](javascript:AssessmentByName('assm-exer-oop-fund');), for example, you created a new Crocodile type.

In this course section, you have learned that information in Java is of a specific kind; we call these kinds of information data types. Java provides certain simple kinds of information, called primitive types. These primitives are the basic building blocks of the Java language. More advanced data types are also available, called reference types. All Java classes are reference types, whether provided in a Java package or created by a programmer.

It is worth noting at this point that the concept of data types is not specific to Java. You will find data types in most other programming languages. You might also have noticed a pattern in identifying primitive and reference types. In Java, primitive types are always lower case nouns and reference types are upper case nouns. This convention makes it much easier for you to read code and you should adhere to it when creating your own classes.

In the next section of the course, you will have the chance to apply your new understanding of data types by learning about variables.

**2.3.1.2 Variables**

**Variables**

In the previous course section, you learned about categorizing information based on the kind of information in question. In Java, this is done by using data types. The ability to store and reuse pieces of information is also very important in programming. Variables comprise the mechanism that programming languages use to store information. In this section of the course, you will learn what variables are and how you can create them.

Variables are a fundamental building block of programming. A variable represents a placeholder or container for information. The concept of variables should not be new to you. In learning algebra, you probably made extensive use of variables. For example, consider the equation 2 + *x* = 5. In this case, *x* represents a variable that has a value of 3. The value of a variable does not need to be constant, however; in different contexts, you can assign different values to variables. In the expression 2 + *x*, the value of *x* is unknown. In this case, you can assign different values to *x* to achieve different results. For example, if *x* is assigned the value of 5, then 2 + *x* = 7; if *x* is assigned a value of 2, then 2 + *x* = 4.

Variables in programming are similar to those that you used in algebra—programming variables can be thought of as a container to store values. In the Chair example discussed in [2.3.1.1 Data Types](javascript:ContentByName('pg-data-types');), recall that the Chair class had four attributes: height, chair color, maximum load, and arms. The values of these attributes may differ for different Chair objects. Indeed, some of these attributes may even change over time — you can adjust the height of an office chair, for example. You can treat each attribute as a variable: height is a container for storing a specific piece of information about the state of a Chair.

Let us consider another situation where variables can be useful. Imagine that:

* We are writing a program.
* Part of the program requires a value to be computed using a very complex computation.
* Because of its complexity, the computation takes one hour to complete, even with a very fast machine. This is a long time, but still much faster than we could perform the calculation ourselves.
* We will have to use the result of this computation in twenty-five different places in our program.

If we need to compute this same value each time we use it, then our program would take twenty-five hours to run to completion. While a one-hour wait for our value was acceptable, a one-day wait probably is not. How can we shorten the time necessary for the program to run?

One obvious solution is to perform the calculation once and then save the result. More specifically, we want to store the value that the computation returns in the computer's memory. This way, though our first computation still requires an hour's time, the 24 remaining uses of the value can make use of the work already performed. This is exactly what variables are designed to do. A variable represents a specific location in a computer's memory where data can be stored and accessed.

Though the above example discusses variables as a way to save time, the need for variables is far more widespread. Most programs are constantly working with values. Users will input new values into a program. Methods will calculate new values that will be needed later. Many calculations will need to store intermediate results for later manipulation. In all of these cases, we need something to store values for later use. Variables perform this role.

**Java is a Strongly Typed Language**

At this point, you know that variables represent a location in a computer's memory used to store and access information. You also have learned that Java distinguishes between different types of information known as data types. If you make a connection between the fact that information in a Java program is of a specific data type and that a variable is a place to store information, you could correctly conclude that variables can be of certain data types.

The reality is not merely that a Java variable *can be* of certain type but every variable in Java *must be* of a certain type.

The reason for this strict use of data types and variables is to ensure proper behavior. If a program accepts a value with an unexpected type, the results can be unpredictable. When you sit down, you normally anticipate sitting on a chair. Imagine that your chair was replaced by another object without your knowledge. The object might be one that you can sit on, such as a table, or it might be an object you cannot sit on, such as a lamp. If you do not know the object in advance, the result of your attempt at sitting is difficult to predict. You could certainly make some contingency plans for certain types of objects, but attempting to make plans for every possible type of object is nearly impossible. Java avoids this problem by only accepting data of the expected type — analogous to your choosing to only sit on chairs in the above example. As you can see, predictability is one reason for strong typing. Just as important, however, is the question of meaning — most operations can only be meaningfully performed on specific data types. Arithmetic operations, for example, can only be performed on numbers, such as int or double. It is meaningless to talk about subtracting a Chair from another Chair.

Strict typing also has an effect on memory use and allocation. As you can see in the chart above, each of the three primitives listed may represent a limited range of information. The only possible values for a boolean are true or false, for example. An int can only be a whole number from (-2,147,483,648) to (+2,147,483,647). The amount of memory allocated for a specific data type limits the range of values that are possible for that data type. A boolean has a size of one bit, which limits its possible values to true or false. An int has 32 bits allocated, which allows for a much larger range of values.

Since a variable represents space in memory, it is important for the JVM to know how much space it needs. By requiring a data type to be specified when declaring a variable, Java ensures that the JVM can allocate enough space for that variable. Since a variable's data type cannot change, we can also be certain that a variable's allocated memory will be sufficient for the life of the variable. In this way, strongly typed languages allow for efficient use of memory.

The importance of dependability in programming cannot be overstated. If a language does not strictly enforce the use of data types, information can be lost or programs can simply fail to function, often with disastrous results. In 1996, the European Space Agency's Ariane 5 rocket exploded less than a minute after lift-off because of a software bug. The rocket's guidance software (which did not use a strongly typed language) expected an integer representing the rocket's horizontal velocity. The velocity information was provided in the form of a floating-point number. The memory required to store the floating point was much larger than the memory allocated to store the integer, resulting in a crash that cost an estimated 7.5 billion US dollars.

Alternatively, imagine a different scenario. If you have a savings account or if you have an outstanding loan, the interest in either case is probably represented by a double. Imagine that your bank could not use a double and instead had to represent all interest as an int. In such a case, the interest that you collect on your savings could be changed from 3.4% to just 3%. The interest on your loan could rise from 7.6% to 8%. While neither of these situations is of the scale of the Ariane 5 explosion, they would probably have a very significant impact on your life!

Java, as a strongly typed language, requires the explicit designation of a data type when declaring a variable. This data type cannot change within the program. Similarly, if a Java program expects to receive data of a specific type, it will not accept data of any other type. This strong typing requires Java programmers to be thorough in their work, but it allows for efficient use of memory and avoids software errors.

**Declaring Variables**

When creating a variable in Java, we must specify a data type and an identifier (a name) for that variable. We refer to this process as "declaring a variable." Declaring a variable will allocate enough space in memory to store the variable's data type. To access information stored in that space later, you need only refer to the identifier. Please note that declaring a variable only allocates space for the variable — it does not actually store any information in that space.

Variable identifiers can be any string that you wish, subject to a few guidelines. Some of these are rules, and Java compiler strictly enforces these rules; others are conventions that make programs easier to read and use for other programmers. Some of these rules and conventions are:

* Rules:
  + Identifiers can contain letters, underscores ( \_ ), or the dollar sign ( $ ).
  + Identifiers cannot start with a number.
  + Identifiers may not contain spaces or other special characters or punctuation marks.
  + Identifiers cannot be a Java keyword.
* Conventions:
  + Variable identifiers should begin with a lower-case letter (a convention).
  + Variable identifiers should be descriptive nouns, adjectives, or noun/adjective phrases (a convention).

Another convention to make identifiers more readable is the capitalization of new words – monthlyInterest or myBankAccount. This is called "camel hump" notation. Based on these rules, the following table shows some valid and invalid identifiers for variables:

|  |  |
| --- | --- |
| **Valid Variable Identifiers** | **Invalid Variable Identifiers** |
| x y personalGreeting \_userID $backgroundImage | double 3sisters State Capitol add@payment |

**Table 1** *Valid and invalid identifiers for variables*

There are many kinds of variables in Java, many of which we will introduce in the next page. The generic declaration of a variable follows the following format

DataType variableName;

For example, the following are valid variable declarations:

int age;

String name;

double salary;

To declare an instance variable in Java, state the access modifier, the data type and the identifier, followed by a semicolon:

private int x;

public boolean continue;

public String personalGreeting;

private HTMLPage indexPage;

## 2.3.1.3 Using Variables

### Assignment

Variables store information for later use; now that you know how to declare variables, it is time to discuss how to put information into a specific variable. We refer to this process as "assigning a value to a variable" and it is most often accomplished using the basic assignment operator. The basic assignment operator is =. To this point in your life, you have probably called this operator an "equals sign" and would have read the statement "x = y" as "x equals y". Within Java, however, the statement "x = y" should be read as "x is assigned the value of y" or "x gets y." It is very important that you note this difference and that you get into the habit of reading "=" as "is assigned the value of" or "gets." Java does have an "is equal to" operator, but its use is very different from that of the basic assignment operator. You will learn about the equality operator later in the course.

To see the structure of an assignment, let us assume that the following declarations have been made:

String myStr;

int x;

int y;

Given those declarations, the following assignments are possible:

myStr = "This is a string";

x = 5;

y = 3;

x = y;

Please be aware that the value you assign to a variable must match that variable's type. You cannot assign an int to a String, for example. As such, the following examples are incorrect and would not compile:

x = "A string";

x = myStr;

myStr = y;

Assignments occur within a method body. Within that constraint, however, variables can be set at any time, as often as needed.

### Initialization

Normally you will assign a value to a variable based upon the result of a calculation. However, there are occasions when you know what initial value the variable should have. In such cases, you may assign a value when you declare the variable. This is known as initialization. It takes the form:

String myStr = "This is a string";

The above statement is equivalent to:

String myStr;

myStr = "This is a string";

In the context of the bank teller example, please look for instance variable initialization in the constructors for BankAccount and AccountsLedger.

### Default Values

If a variable does not have a value assigned, Java helps by initializing the variable to a default value. In the case of primitive data types, the JVM initializes the variable to whatever value is specified as the default for that type (refer to the Java API's discussion of primitive data types to see these default values).

Reference variables are initialized to null. As you will recall from our earlier discussion in [2.2.3 Lifecycle of Objects](javascript:ContentByName('pg-objects-lifecycle');), if a reference variable has the value of null, then the variable does not refer to any object.

### Scope

"The scope of a declaration is the region of the program within which the variable declared by the declaration can be referred to using a simple name" ([Java Language Specification](http://java.sun.com/docs/books/jls/second_edition/html/names.doc.html" \l "103228" \t "externalWindow)). If you attempt to refer to a variable using its simple name in a location that is outside its scope, your program will not compile. If it does compile, your program will produce unexpected results because your reference refers to some other variable of the same name and not the one you intended. As we will see later, we can refer to a variable outside its scope by using a qualified name.

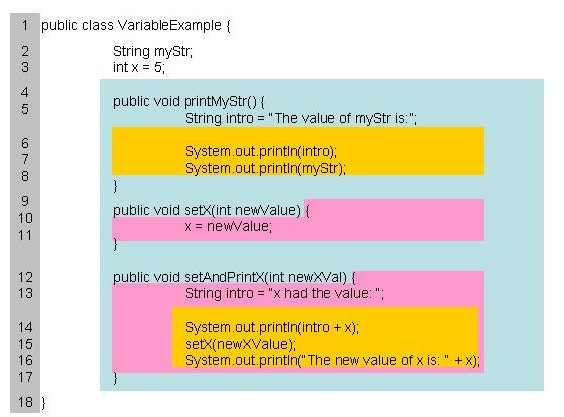
Member variables are declared at the class level, outside of any method or constructor. A member variable's scope is the entire class in which it has been declared, except when shadowed by a local variable declaration. We will discuss shadowing a little later. A member variable is either an instance variable or a class variable.

Local variables are usually used to store the intermediate value of some computation in a block of code. A pair of braces delimit a code block. A code block consists of local variable declarations and a sequence of statements to be executed by the computer. A local variable's scope is from its point of declaration to the end of its code block. This kind of variable is accessible in the code block, but inaccessible outside the block.

Parameters are passed into a code block as arguments to methods or constructors. A parameter's scope is the entire method or constructor for which it is a parameter.

The example below shows member variables, local variables, and parameters:

Figure 1 Scope of member variables, local variables, and parameters



In this example, the class VariableExample has two member variables: myStr and x. The scope of these variables is the entire class and is represented by the blue area (from line 4 to line 17) — they can be accessed and/or modified anywhere in the blue area. Note that x has an initial value set at declaration and the setX method modifies the value of x. Similarly, x is used in the setAndPrintX method.

There are also two local variables; both of these variables have the identifier intro. Despite having the same name, these are two separate variables — they are not visible outside of their respective methods (printMyStr and setAndPrintX). Scope for these variables is represented by the orange area (from line 6 to line 7 and from line 14 to line 16).

Finally, there are two parameters: newValue and newXVal. newValue is available throughout the pink area (from line 9 to line 10). The scope of newXVal is also represented by the pink coloring, although within the setAndPrintX method (from line 12 to line 16).

### Lifetime

In SSD1, we are concerned with four kinds of variables: class variables, instance variables, local variables, and parameters. The time at which JVM allocates memory for a variable depends on the kind of variable. The time at which JVM recovers the allocated memory also depends on the kind of variable.

#### Class Variable

Recall from our discussion of class attributes in [2.2.1 Designing Classes](javascript:ContentByName('pg-design-class');) that a class attribute is common to all instances of the class and that there is only one copy of a class variable regardless of the number of object instances. A class variable comes into existence even before the first object of that class is created. Therefore, the JVM allocates memory for a class variable right after loading a class. For example, the class variable nAlgaeCreated in the AlgaeColony class comes into existence right after the JVM successfully loads the AlgaeColony class.

The JVM recovers the memory allocated for class variables right after the class is unloaded. In the iCarnegie Servlet Workbench, if you right-click on a .class file in the Java-Classes folder of the workbench, and click on **Delete** in the drop-down menu that appears, the class is unloaded.

#### Instance Variable

The JVM allocates memory for an instance variable just before executing the constructor code and recovers the memory when garbage collecting the object instance. Thus, the JVM creates the instance variable energy for AlgaeColony right before executing the constructor.

#### Local Variable

The JVM allocates memory for a local variable before executing the block of code in which the local variable is declared. The JVM recovers the memory after executing the block.

#### Parameters

JVM allocates memory for a parameter before the method or constructor starts executing. JVM recovers the memory allocated to parameters when the message processing is complete.

### Name

Each member variable, constant, method, and class has a name (identifier). When using a name, we can use either a simple name or a qualified name.

#### Simple Name

A simple name consists solely of the identifier. For example, the getEnergy method of AlgaeColony has to return the current energy level of the AlgaeColony object. We will use a simple name of the instance variable energy to return the value:

|  |  |
| --- | --- |
| 1:  2:  3:  4:  5:  6:  7:  8: | /\*\*  \* Get the energy currently carried by the algae.  \*  \* @return current energy level of the organism  \*/  public int getEnergy() {  return energy;  } |
| Listing 1 Simple name | |

Simple names suffice in most situations, since the entities (variables, methods, classes) we want to refer to are often within scope. Sometimes, we need to refer to an entity and have no way of referring to that entity using a simple name because we are outside its scope. In such situations, we use a qualified name to specify the context in which the entity resides.

#### Qualified Name

A qualified name has one or more qualifying identifiers preceding the name. A qualified name consists of a sequence of qualifying identifiers separated by periods or dots (.). The qualifying identifiers provide the entity's context. We typically use a qualified name only to refer to entities (variable, method, or class) that are out of scope. For example, the getDisplayMechanism method in AlgaeColony uses the qualified name of the class constant COLOR. The constant is declared in the Simulation class:

|  |  |
| --- | --- |
| 1:  2:  3:  4:  5:  6:  7:  8: | /\*\*  \* The mechanism to display Algae is to use its color.  \*  \* @return a constant defined in {@link Simulation#COLOR Simulation} class  \*/  public String getDisplayMechanism() {  return Simulation.COLOR;  } |
| Listing 2 Qualified name | |

You can use a qualified name where a simple name would suffice (i.e., even if the entity is in scope). A common example is the use of the keyword this — a reference to an object. In the getEnergy example discussed above, we could have used this.energy in place of energy. In the body of a method, the keyword this denotes a reference to the object for which the method was invoked.

### Summary

In this section of the course, you learned about data types and variables in Java. Data types can be broadly categorized into two different groups: primitive types and reference types. Primitive types are the simplest kinds of information in Java and they form the foundation for all Java programs. Some examples of primitive types are boolean, char, double and int. Reference types are any pieces of information that are represented by a Java class.

Java variables represent a location in memory that is allocated for storing information. Because Java is a strongly typed language, all variables have an associated, immutable data type. This data type must be specified when a variable is declared. The scope of a variable limits where the variable can be referred to by its simple name. A variable declared at the class level is known as a member variable; its scope is the entire class in which it was declared. Local variables are declared within the body of a method or constructor. Local variables can be referenced only within the code block in which they were declared. Parameters are declared as arguments to a method or a constructor; their scope is limited to the method or constructor for which they are an argument. Attempting to reference a variable outside of its scope will cause an error during compilation.

Java variables are assigned values using the assignment operator. Initial values may be set when a variable is declared; this is known as initialization. After declaration, a variable may have a value assigned anywhere within its scope, as often as is necessary. Note a constant may be assigned a value only once. Constants are called final fields in Java.

## 2.3.2 Arithmetic Operators and Expressions

In the last section of the course, you learned how to store information in variables. Information storage is certainly important, but it is not enough to make a program. The ability to evaluate and manipulate the information you have stored is necessary for you to make effective use of variables.

This manipulation and evaluation is done by means of *operators* and *expressions*. An expression contains one or more values that are manipulated ("operated upon") by one or more operators. To offer a simple example, 2 + 3 is an expression. It contains two values (2 and 3) which are manipulated by the addition operator. All expressions evaluate to a value; in the example, 2 + 2 evaluates to 4. It is important to realize that in most cases in Java an expression can be used interchangeably with the value to which it evaluates. This means that the following two lines of code are equivalent:

int x = 4;

int x = 2 + 2;

**Arithmetic Operators**

The arithmetic operators should be familiar to you, since you have been using them since you first learned basic arithmetic. The following examples illustrate the integer arithmetic that can be done in Java with different operators. For all these examples, assume x always has a value of 100 as given by the declaration:

int x = 100;

|  |  |  |  |
| --- | --- | --- | --- |
| **Operation** | **Operator** | **Example** | **Evaluates to:** |
| Addition | + | x + 5 | 105 |
| Subtraction | - | x - 5 | 95 |
| Multiplication | \* | x \* 2 | 200 |
| Division for Quotient | / | x / 2 | 50 |
| Division for Remainder | % | x % 2 | 0 |

**Table 1** *Examples of arithmetic operations in Java*

In the context of the bank teller example, look at the implementations of withdraw and deposit methods of the BankAccount class for examples of arithmetic operations.

**Using Arithmetic Operators and Expressions**

The SimpleCalculator class, detailed below, uses the HTML file [SimpleCalculatorForm.html](http://www.icarnegie.com/content/SSD/SSD1/2.1/normal/pg-intro-java-oop/pg-java-fund/pg-arithmetic-operators/SimpleCalculatorForm.html" \t "externalWindow) to submit input. Lines 26-30, offers an example of each of the five arithmetic operators that you'll be expected to use in this course.

|  |  |
| --- | --- |
| 1:  2:  3:  4:  5:  6:  7:  8:  9: 10: 11: 12: 13: 14: 15: 16: 17: 18: 19: 20: 21: 22: 23: 24: 25: 26: 27: 28: 29: 30: 31: 32: 33: 34: 35: 36: 37: 38: 39: 40: 41: 42: 43: 44: 45: 46: 47: 48: 49: 50: 51: 52: 53: | import java.io.\*;  import javax.servlet.\*;  import javax.servlet.http.\*;  public class SimpleCalculator extends HttpServlet {  public void doPost(  HttpServletRequest request,  HttpServletResponse response)  throws ServletException, IOException {  int input1;  int input2;  int additionResult;  int subtractionResult;  int divisionResult;  int multiplicationResult;  int modResult;  // Get the user input from the form via request object  // and assign those values to local variables  input1 = Integer.parseInt(request.getParameter("input1"));  input2 = Integer.parseInt(request.getParameter("input2"));  // perform calculations and assign  // results to relevant local variables  additionResult = input1 + input2;  subtractionResult = input1 - input2;  divisionResult = input1 / input2;  multiplicationResult = input1 \* input2;  modResult = input1 % input2;  response.setContentType("text/html");  PrintWriter out = response.getWriter();  // Write the response  out.println("<!DOCTYPE HTML PUBLIC '-//W3C//DTD "  + "HTML 4.01 Transitional//EN'>");  out.println("<html><head><title> Calculation Result "  + "</title></head><body>");  out.println("Your Calculations:");  out.println("<BR>" + input1 + " + " + input2  + " = " + additionResult);  out.println("<BR>" + input1 + " - " + input2  + " = " + subtractionResult);  out.println("<BR>" + input1 + " / " + input2  + " = " + divisionResult);  out.println("<BR>" + input1 + " \* " + input2  + " = " + multiplicationResult);  out.println("<BR>" + input1 + " % " + input2  + " = " + modResult);  out.println("</body></html>");  }  } |
| Listing 1 *[SimpleCalculator](http://www.icarnegie.com/content/SSD/SSD1/2.1/normal/pg-intro-java-oop/pg-java-fund/pg-arithmetic-operators/SimpleCalculator.java" \t "external)* | |

#### AlgaeColony

Let us briefly discuss the use of arithmetic operators in AlgaeColony by coding the liveALittle method. An algae needs to do two things while living:

1. Grow older (i.e., age).
2. Convert energy from sunlight into life-energy.

To grow older, we need to increment the value of instance variable age.

Regarding energy conversion, algae will convert half the available solar energy into life energy. How would you find the amount of sunlight available at the algae's location? Hint: The Simulation class provides a service that we can use for this purpose. Read its [documentation](http://www.icarnegie.com/content/SSD/SSD1/2.1/normal/pg-simulation/Alife/doc/Simulation.html" \t "externalWindow).

Once we get the amount of sunlight from Simulation, we will multiply that by half and add it to the current energy level less the amount of energy needed to live. Putting this all together, we have the implementation for liveALittle:

|  |  |
| --- | --- |
| 1:  2:  3:  4:  5:  6:  7:  8:  9: 10: 11: 12: 13: 14: 15: 16: 17: 18: 19: | /\*\*  \* Algae lives its life. May gain or lose energy.  \*/  public void liveALittle() {  if (isDead()) {  return;  }  age = age + 1;  int row = getRow();  int column = getColumn();  int sun = simulation.getSunlight(row, column);  // Let us assume that algae can convert 50% of  // solar energy into life-energy.  // Algae has to give up some energy to live.  setEnergy((int) (sun \* 0.5) + getEnergy() - ENERGY\_TO\_LIVE);  } |
| Listing 2 method *liveALittle* of *AlgaeColony* | |

### OOP Practice: Use an Accessor when Available

Note that the example above used the "getter" methods for row and column. It is good OOP practice to use getter methods (also called accessor methods or accessors) wherever available instead of directly using an instance variable's value: If we have to change how we read the value of the instance variable in the future, it is easier to do so if all the reading happens through the accessor method. We would need to change only the accessor method. This is much simpler than changing each line of code that uses the instance variable.

For example, let us say that AlgaeColony.java has ten snippets of code that use row and column instance variables, all dispersed at various places. In the future, if we decide to use a separate Location class to represent row and column location, we would need to change all ten snippets of code. On the other hand, if we had used the accessor methods, we would need to change only two snippets of code: the two accessor methods.

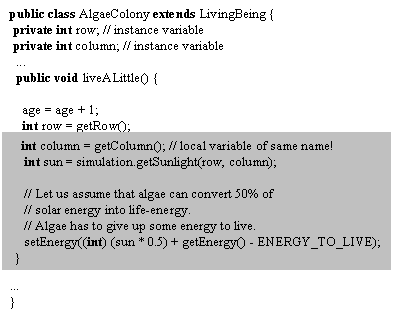
### Casting

In the liveALittle method above, we multiply the sunlight value by 0.5. The resulting number will be a double. However, the setEnergy method expects an int. We need to convert the double to an int. To convert a numeric value of one type to a similar value of another numeric type at run time, Java provides casting. To cast, specify within parentheses the type you want to cast to. The liveALittle example above shows how we can cast the type of a numeric value (double) to another primitive type (int). Note that this kind of cast truncates fractional information — it does not round to the nearest integer. Later in this course, we will use casting to convert the reference type of a reference to another reference type.

### Shadowing

In the liveALittle method, we declared column as a local variable. Recall that AlgaeColony has an instance variable of the same name. Which one do we reference when using the simple name column? When we declare a variable with the same name as another variable that is already in scope, the new declaration is said to shadow the declaration of the variable that was already in scope. Shadowing continues until the new declaration goes out of scope. Thus, any reference to column in the grey area in the example below will refer to the local variable, and any reference to column in the white area will refer to the instance variable.

Figure 1 Local variable shadows an instance variable



Note the same shadowing effect applies to the variable named row as well. To refer to a variable that is under shadow, use a qualified name. The qualified name for the instance variable column is this.column, which may be used to refer to the instance variable in the shadow region.

### Increment Operator

In the liveALittle method, we incremented the value stored in the instance variable age by assigning a new value. The new value was computed by adding one to the value stored in age. Thus, we performed two operations: addition and assignment. There is a shortcut in Java to increment the numeric value stored in a variable by one, the ++ operator, called the increment operator. The two statements below are equivalent:

* age = age + 1;
* ++age;

We read the second statement as "Increment age."

In this page, we introduced arithmetic operators and expressions. Boolean operators are another class of operators in programming languages. We will cover them in the next page.

**2.3.3 Boolean and Relational Operators and Expressions**

In the last section of the course you learned about arithmetic operators and expressions; these expressions evaluate to a number value. Obviously, the ability to perform mathematical operations is important in programming. However, it is equally important to evaluate certain conditions: Imagine an online ordering system that offers certain discounts – a 10% discount for students under age 24 and free shipping for orders over $50.00. In such cases, you would need to evaluate whether the order meets these criteria: Is the customer both a student and under age 24? Is the total order greater than $50.00? Such conditions have only two possible values: true or false. Expressions that represent these types of conditions are called "Boolean" expressions. There are a number of special operators in Java to build these Boolean expressions.

Many students find it difficult to understand why or how one would use Boolean expressions initially; if you find yourself in this situation, you are not alone. In the next section of the course (control structures), the use of Boolean expressions should become clearer. In the meantime, focus on learning the operators below.

**Relational Operators**

Very often, we need to compare two entities (numeric values, reference values, Boolean values). This is done using relational operators, such as "greater than" and "less than." The table below shows some common Java relational operators:

|  |  |
| --- | --- |
| **Operator** | **Symbol** |
| Greater than | > |
| Less than | < |
| Greater than or equal to | >= |
| Less than or equal to | <= |
| Exactly equal to | == |
| Not equal to | != |
| instance of | instanceof |

**Table 1** *Relational operators in Java*

Let us look at an example of relational operator usage. In our AlgaeColony class, the isDead method should return a Boolean value of true if the algae are dead and false if it is alive:

|  |  |
| --- | --- |
| 1:  2:  3:  4:  5:  6:  7:  8: | /\*\*  \* Is the algae dead?  \*  \* @return <code>true</code> if dead. <code>false</code>, otherwise.  \*/  public boolean isDead() {  return (deadOrAlive == DEAD);  } |
| Listing 1 *Relational Operator* | |

**Boolean Operators**

You typically use Boolean operators when you have two or more Boolean expressions and you want to combine them to get a new Boolean expression. There are many examples of Boolean operators and expressions in the real world. For this course, you need to know only about the following operators.

|  |  |
| --- | --- |
| **Operator** | **Symbol** |
| AND | && |
| OR | || |
| NOT | ! |

**Table 2** *Boolean operators in Java*

**AND Operator**

The Boolean AND expression evaluates to true only when both the component Boolean expressions evaluate to true. To return to our earlier example: If the customer is a student AND the customer is younger than 24, the customer receives a 10% discount. Here, the following two Boolean expressions are combined to yield a third expression:

1. Is the customer a student?
2. Is the age of the customer less than 24?

Receives discount: (Is the customer a student) && (Is the age of the customer less than 24)

Only when both of the above Boolean expressions are true will the entire Boolean expression evaluate to true. The customer receives a discount only when the Boolean expression is true.

**OR Operator**

A Boolean OR expression evaluates to true only when either or both of the component Boolean expressions evaluate to true. Here is an example of a Boolean OR: If it is raining or snowing outside, you cannot play. An OR joins two Boolean expressions together operator to form a third Boolean expression:

1. Is it raining?
2. Is it snowing?

Cannot play: (Is it raining) || (Is it snowing)

If either of them evaluates to true, the entire Boolean expression evaluates to true. When the Boolean expression evaluates to true, you cannot play outside.

**NOT Operator**

Here is an example of a NOT operator: If your height is NOT less than five feet you can join the police force.

can join: !(height less than five feet)

**Complex Boolean Operators**

Here are some examples of complex Boolean operators:

* If the reservoir is NOT empty AND the water heater is working, you can take a hot shower.
* If the temperature is hotter than 95 degrees Fahrenheit OR lesser than 60 degrees Fahrenheit AND you do not have air conditioning AND heater, it will get uncomfortable.

**Boolean Type and Boolean Variables**

Boolean is a type just like int and double. Just as you can have variables of type int and double, you can also have variables of type boolean. Boolean variables can take only one of two values: true or false. Since a Boolean expression evaluates to a Boolean value, you can assign any Boolean expression to a variable. In the example below, we make use of boolean variables. The following table illustrates the result of the three Boolean operators when two boolean variables take on different values.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **a** | **b** | **a && b** | **a || b** | **!a** |
| true | true | true | true | false |
| true | false | false | true | false |
| false | true | false | true | true |
| false | false | false | false | true |

**Table 3** *Truth-table of AND, OR, and NOT*

**Rules for Evaluating a Boolean Operator**

Notice that the Boolean operator && (AND) and || (OR) require two operands, while the ! (NOT) operator requires only one operand. When you use the Boolean AND operator, and the first operand evaluates to false, the second operand will not be evaluated. Irrespective of the value of the second operand, the AND expression will evaluate to false. Let us take the example: "If your height is five feet or higher AND you can run the 100 meters race in 11 seconds, you can join the police force." When the first condition, height is five feet or higher, evaluates to false, there is no need to evaluate the second condition. You cannot join the police force.

Similarly, for the OR operator, the second operand is not evaluated when the first operand evaluates to true. Let us consider the example: If it is raining or snowing outside, you cannot play. When the first condition: "Is it raining outside" evaluates to true, there is no need to check for the second condition (even if it could both rain and snow at the same time). You cannot play.

As we mentioned earlier, it is difficult to see the usefulness of some of these operators and expressions without some context. We will be providing that context in the next section of the course.

**2.3.4 Control Flow**

As illustrated by some of the examples in [2.3.3 Boolean and Relational Operators and Expressions](javascript:ContentByName('pg-boolean-and-relational-operators');), a program may need to do different things depending upon certain conditions. Returning to our ordering example, the program needs some way to evaluate whether or not a customer is both a student and under age 24, in order to apply a discount based on the result of this evaluation. This example illustrates the need to control the flow of a program. In this section, you will learn to control program flow based on conditions.

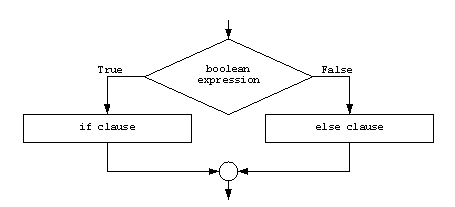
**Sequential Execution and Control Structures**

A computer executes statements in a code block sequentially in the order in which the statements appear. Many times, we may need the order of execution to be different from the order in which they appear. To enable a programmer to control the order of execution, most programming languages provide several options. These options are collectively called **control structures**.

**If-Else-Statement**

We often use flowcharts to illustrate control structures. A flowchart is a graphical representation of the order in which statements are executed. We use symbols such as diamonds, rectangles, and circles to draw flowcharts. A diamond is used when a test is to be evaluated to decide which way to proceed. Use a rectangle to denote sequential execution of statements in that rectangle. Lines connect these symbols with arrows called flow-lines. Execution proceeds in the direction of the arrow. Use a circle as a junction point where multiple flow-lines meet. A flow-chart of an if-else-statement is given below.

**Figure 1** *Flow-chart of an if-else-statement*



The syntax of an if-else-statement is:

if (booleanExpression) statement

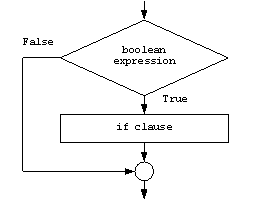
else statement

In the above, the statement can be a single statement, a block of statements (enclosed by curly braces) or a null statement (just a semicolon). Thus, all of the following are valid:

* if (isRaining) canPlay = false; // A single statement
* if (isRaining); // Null statement. Does nothing.
* if (isRaining) {
* canPlay = false;
* takeUmbrella = true;
* } // block of statements
* if (order > 50.00) {
* freeShipping = true;
* } else {
* freeShipping = false;
* }

If there is no else-clause, the flowchart looks like the following:

**Figure 2** *Flow-chart of an if-statement*



In the context of the bank teller example, please look at the implementation of deleteAccount method of AccountsLedger for an example of the if-statement.

Another example of the if-statement is shown in the example below.

**Example**

The PriceCalculator class, detailed below, uses the HTML file [PriceCalculatorForm.html](http://www.icarnegie.com/content/SSD/SSD1/2.1/normal/pg-intro-java-oop/pg-java-fund/pg-controlflow/PriceCalculatorForm.html" \t "externalWindow) to submit input. Lines 36 to 42 detail the use of an if-statement.

|  |  |
| --- | --- |
| 1:  2:  3:  4:  5:  6:  7:  8:  9: 10: 11: 12: 13: 14: 15: 16: 17: 18: 19: 20: 21: 22: 23: 24: 25: 26: 27: 28: 29: 30: 31: 32: 33: 34: 35: 36: 37: 38: 39: 40: 41: 42: 43: 44: 45: 46: 47: 48: 49: 50: 51: 52: 53: 54: 55: 56: 57: 58: 59: 60: 61: 62: 63: 64: 65: 66: 67: 68: 69: 70: 71: 72: | import java.io.\*;  import javax.servlet.\*;  import javax.servlet.http.\*;  public class PriceCalculator extends HttpServlet {  public void doPost(  HttpServletRequest request,  HttpServletResponse response)  throws ServletException, IOException {  boolean isStudent;  int age;  double order\_price;  double discount;  double shipping = 10;  double total\_price;  String student;  String orderPrice;  response.setContentType("text/html");  PrintWriter out = response.getWriter();  /\*\*  \* Set the appropriate variables  \*/  student = request.getParameter("student");  orderPrice = request.getParameter("order\_price");  isStudent = Boolean.valueOf(student).booleanValue();  age = Integer.parseInt(request.getParameter("age"));  order\_price = Double.parseDouble(orderPrice);  /\*  \* calculateDiscount snippet  \* Permits order to have only one discount applied.  \* An order may receive a 10% discount if order  \* is under 24 and a student.  \* Orders over $50.00 that do not recieve the above  \* discount qualify for free shipping.  \* In all other cases the order does not qualify  \* for a discount.  \*/  if ((isStudent) && (age <= 24)) {  discount = order\_price \* 0.1;  } else if (order\_price > 50.00) {  discount = shipping;  } else {  discount = 0;  }  /\*  \* Calculate final order price  \*/  total\_price = order\_price + shipping - discount;  /\*  \* Write the response  \*/  out.println("<!DOCTYPE HTML PUBLIC '-//W3C//DTD "  + "HTML 4.01 Transitional//EN'>");  out.println("<html><head><title> Calculation Result"  + "</title></head><body>");  out.println("Order Price: " + order\_price);  out.println("<BR>Shipping: +" + shipping);  out.println("<BR>Discount: -" + discount);  out.println("<HR width='150' align='left'>Total Price: +"  + total\_price);  out.println("</body></html>");  }  } |
| Listing 1 *[PriceCalculator](http://www.icarnegie.com/content/SSD/SSD1/2.1/normal/pg-intro-java-oop/pg-java-fund/pg-controlflow/PriceCalculator.java" \t "external)* | |

The calculateDiscount snippet (lines 36-42) offers one implementation of the discount specification discussed above. The program will first check to see if the order qualifies for the first discount (i.e., is this a student and is the age less than or equal to 24). If the conditions are met, the 10% discount is applied and no further action is taken.

If the conditions for the first discount are not met, then the program will continue through the if-else-statement and check for the second possible discount: is the order price more than $50.00? If so, the free shipping discount is applied.

If neither of the two discount conditions are met, no discount is applied (hence, discount is set to 0 on line 41).

As currently written, the calculateDiscount snippet will allow only one of two possible discounts for an order. If an order qualifies for the student discount, it cannot qualify for the free shipping discount. If the specifications for the discount were different, we would have to adapt our if-else-statement accordingly. For example, if an order could qualify for both discounts at once, we could write our calculateDiscount snippet as follows:

|  |  |
| --- | --- |
| 1:  2:  3:  4:  5:  6:  7:  8:  9: 10: 11: 12: | /\*  \* Calculate discount  \*/  discount = 0;  if ((isStudent) && (age <= 24)) {  discount = discount + order\_price \* 0.1;  }  if (order\_price > 50.00) {  discount = discount + shipping;  } |
| Listing 2 *[Multiple discounts](http://www.icarnegie.com/content/SSD/SSD1/2.1/normal/pg-intro-java-oop/pg-java-fund/pg-controlflow/calculateDiscount2.java" \t "external)* | |

To give you one more look at arranging conditional statements, imagine that an order may receive either free shipping or the student discount, whichever is greater. The following simple if statements accomplish this:

|  |  |
| --- | --- |
| 1:  2:  3:  4:  5:  6:  7:  8:  9: 10: 11: 12: | /\*  \* Calculate discount  \*/  discount = 0;  if ((isStudent) && (age <= 24)) {  discount = order\_price \* 0.1;  }  if ((order\_price > 50.00) && (shipping > discount)) {  discount = shipping;  } |
| Listing 3 *[Greater of two discounts](http://www.icarnegie.com/content/SSD/SSD1/2.1/normal/pg-intro-java-oop/pg-java-fund/pg-controlflow/calculateDiscount3.java" \t "external)* | |

**Nested If-Statements**

In many cases, a single if-statement will not adequately meet your needs. You may need to further control the flow of a program based upon earlier conditions. In such cases, you can actually place an if-statement inside the body of another if-statement. We refer to this structure as "nesting." Nesting is quite common in control structures. Let us return to our calculateDiscount snippet and make the criteria for a discount more complex. In our new example, we modify the student discount so that orders that are $100.00 or less receive a 10% discount, while orders that are greater than $100.00 receive a 15% discount. In this new example, the program first checks to see if the criteria for the student discount are met. If the order qualifies for the student discount, then the conditions for the discount are examined. Also note that we have modified the code to check for the shipping discount only if the order does not qualify for the student discount (i.e., an order cannot receive both discounts):

|  |  |
| --- | --- |
| 1:  2:  3:  4:  5:  6:  7:  8:  9: 10: 11: 12: | /\*  \* Calculate discount  \*/  if ((isStudent) && (age <= 24)) {  if (order\_price <= 100.00) {  discount = order\_price \* 0.1;  } else if (order\_price > 100.00) {  discount = order\_price \* 0.15;  }  } else if (order\_price > 50.00) {  discount = discount + shipping;  } |
| Listing 4 *[Nested if-statement](http://www.icarnegie.com/content/SSD/SSD1/2.1/normal/pg-intro-java-oop/pg-java-fund/pg-controlflow/calculateDiscount4.java" \t "external)* | |

**Implementing Catfish**

Now that you have learned about operators and control structures, we can return to the Catfish class and implement some of the methods that we identified in [2.2 Fundamentals of Object-Oriented Programming](javascript:ContentByName('pg-oop-fund');).

First, let us examine the relatively simple getImage method. As we detailed in our specification, the method should return a String representing the filename of the Catfish image. Since our Catfish can face in different directions, the method will need to examine the current value of the direction variable and return an appropriate value. In this case, we check for each of the four possible directions, one at a time, and return the appropriate image for each direction. In addition, we return a default String in case none of the four conditions is met. This default is necessary because without it there is a possibility that none of the four conditions will be met. In such a case, the method would not have a valid return statement, which means that the program will not compile. For each of the conditions, we want to compare the current direction of Catfish (using the accessor method getDirection) with the direction constants declared in Catfish (RIGHT, LEFT, etc.):

|  |  |
| --- | --- |
| 1:  2:  3:  4:  5:  6:  7:  8:  9: 10: 11: 12: 13: 14: 15: 16: 17: 18: 19: 20: | /\*\*  \* @return filename of Catfish image  \*/  public String getImage() {  if (getDirection() == RIGHT) {  return "/Catfish-right.gif";  }  if (getDirection() == LEFT) {  return "/Catfish-left.gif";  }  if (getDirection() == UP) {  return "/Catfish-up.gif";  }  if (getDirection() == DOWN) {  return "/Catfish-down.gif";  }  return "/Catfish-right.gif";  } |
| Listing 5 *[getImage](http://www.icarnegie.com/content/SSD/SSD1/2.1/normal/pg-intro-java-oop/pg-java-fund/pg-controlflow/getImage.java" \t "external)* | |

Now that we have seen a simple method, we move on to one that is a bit more complicated: the moveToRow method. According to our specification, when called by a Catfish object that is alive, moveToRow should change the row of the Catfish to the position specified by the parameter newRow. There is one constraint to this movement, however: the Catfish cannot move beyond the boundary of the lake. In addition, the direction that the Catfish faces should be modified to reflect whether it has moved up or down. Drawing from this specification, we can specify the following sequence of events for the moveToRow method:

1. Check to see if the Catfish is dead. If it is dead, change its position to remove it from the lake. Otherwise, continue.
2. Check to see if the new position (newRow) is within the boundary of the lake. In this case, the boundaries are the top of the lake (row 1) and the bottom (the maxRow of the simulation). If the newRow would be outside of the lake, move the Catfish to the boundary and stop there by changing newRow to have a boundary value.
3. If the Catfish has moved up to get to the newRow, change its direction to UP. If it has moved down, change its direction to DOWN.
4. Set the value of row to newRow.
5. Return the new value of row.

Having specified our event sequence, we can now implement the code for each of the steps.

1. We will first check to see if the Catfish is dead; this can be done using the isDead method discussed in our earlier specification. If isDead returns true, then the method should return a value of -1.
2. if (isDead()) {
3. return -1;

}

1. Next, we need to create code that prevents the Catfish from moving beyond the lake boundaries. In this case, we need to check for two conditions — moving beyond the top of the lake and moving beyond the bottom of the lake. To accomplish this, we will use an if-else-if-statement. We will first check to see if newRow is greater than the maxRow, using the method simulation.getMaxRow. If newRow is greater, then we need to change the value of newRow to the edge of the lake. Next, we will check for the bottom of the lake. If newRow is less than 1, the value of newRow will be set to 1. This can be done using the following code:
2. // Keep the new value within lake boundary.
3. if (newRow > simulation.getMaxRow()) {
4. newRow = simulation.getMaxRow();
5. } else if (newRow < 1) {
6. newRow = 1;

}

1. We now need to examine the direction that the fish has moved and adjust the value of the direction variable accordingly. Again, we have two possible conditions: either the value newRow is less than the current row value (indicating a direction of UP), or newRow is greater than row (indicating a DOWN direction). If neither is true, then the direction the catfish faces should remain the same.

// I might face a new direction  
if (newRow < row) {  
     direction = UP;  
}  
else if (newRow > row) {  
     direction = DOWN;  
}

1. Having taken into account the possibility of the Catfish attempting to move beyond the boundary of the lake and adjusted the value of newRow as appropriate, we can now safely set the value of row to reflect the new position of the Catfish as represented by newRow:

row = newRow;

1. Finally, we can return the value of row:

return row;

After adding some documentation, our completed method is shown below:

|  |  |
| --- | --- |
| 1:  2:  3:  4:  5:  6:  7:  8:  9: 10: 11: 12: 13: 14: 15: 16: 17: 18: 19: 20: 21: 22: 23: 24: 25: 26: 27: 28: 29: 30: | /\*\*  \* Move the catfish to a new row,  \* if new row is within lake bounds.  \*  \* @param newRow - the row to move to.  \* @return the row moved to. Lake boundary limits movement.  \*/  private int moveToRow(int newRow) {  if (isDead()) {  return -1;  }  // Keep the new value within lake boundary.  if (newRow > simulation.getMaxRow()) {  newRow = simulation.getMaxRow();  } else if (newRow < 1) {  newRow = 1;  }  // I might face a new direction.  if (newRow < row) {  direction = UP;  } else if (newRow > row) {  direction = DOWN;  }  row = newRow;  return row;  } |
| Listing 6 *[moveToRow](http://www.icarnegie.com/content/SSD/SSD1/2.1/normal/pg-intro-java-oop/pg-java-fund/pg-controlflow/moveToRow.java" \t "external)* | |

**Student Activity**

In the course section on arithmetic operators, you were given the code to create a simple calculator ([SimpleCalculator.java](http://www.icarnegie.com/content/SSD/SSD1/2.1/normal/pg-intro-java-oop/pg-java-fund/pg-arithmetic-operators/SimpleCalculator.java)). The calculator allowed the user to enter two values (via two text fields) and displayed five expressions using the five operators that you have learned. Modify this program so that, in addition to entering two values, the user must also select one specific operator (+, -, \*, / or %). When the user submits this information, your program should display the result of the expression that he or she has specified.

## 2.3.5 Iteration

### The Need for Iteration

We have worked through some interesting programming problems. Thus far, we have seen two "forms" of code execution in our programs. We started with a style of execution called sequential execution. Sequential execution means that all of the statements in our programs are executed one after another, starting at the top and moving in sequence down to the bottom of the list of statements. No statements are skipped and none is executed more than one time. Therefore, if we are working with a specific method, say the println method, and we want to print 37 String instances, we have to write 37 lines of code calling that method.

Later we worked with what we call control structures, which allow *selective execution* — that is, they allow execution to select different paths (or branches) through our Java statements — and we explored and used the if statement and the if-else statement. (Note that some authors consider these statements the same control structure and other authors consider them two different control statements; it does not really matter to Java as long as you use them correctly.)

The if-statement allows our program to evaluate a conditional expression to determine if they are true or false. If the expression evaluates to true, then a series of statements is executed. If the expression is false, the series of statements is skipped.

The if-else statement is similar — if the expression evaluates to true, statements are executed — just as with the if statement — but if it is false, execution passes to the alternative statement or series of statements that follow the keyword else.

You may want to refer back to the course sections on Boolean expressions and control structures before continuing.

So, we have seen two forms of execution: sequential and selective. This module introduces you to the last form of execution: iteration or repetitive execution.

The shortcoming of selective execution is that it does not allow us to repeat similar tasks easily. Those 37 Strings still require 37 statements, even if we are very clever with one of the control structures. *Iteration*, on the other hand, allows the program to repeat statements until some condition is met. You will learn in this module that Java offers three forms of iteration for the programmer to use (although we cover only two in detail). With these forms of iteration or *loops*, programmers can create applications that repeat one or more Java statements as many times as required.

### Loops

The Java language offers three different loops. All of these loops do essentially the same thing – they start and continue iterating over a set of instructions as long as the Boolean expressions that control them (the loops) evaluate to true, and once the expressions evaluate to false, the loops "terminate" and the iterations cease. However, although these loops do essentially the same thing, having a variety of forms allows us to choose among options when writing code for particular tasks. It is desirable, and usually possible, to make the structure of your program match the intent of your program. As you continue in the course, you will learn more about the different options various loops provide, and you will learn how to choose the most effective loop for your program. In this section, you will learn the while loop:

### The While-Loop

|  |  |
| --- | --- |
| **Form of a while loop** | **Diagrammatic representation of how the loop is executed by the computer.** |
| while (loop guard expression)  loopBody |  |

Table 1 Form of a *while* loop

Here is a simple example of a while loop and the output it generates (assume that these statements are in the doPost method).

int i;   
i = 1;   
while (i < 5) {   
      out.println("<br>value of i is " + i);   
      i = i+1;  
}

The output would be:

value of i is 1

value of i is 2

value of i is 3

value of i is 4

The code within the parentheses of the loop is called the loop guard:

(i < 5)

This expression is evaluated to see if the loop continues to iterate or stops. If this expression evaluates to true, the loop continues the iteration. If this expression is false, the loop terminates iteration and execution continues with the statement following the loop. The guard can be any reasonable Boolean expression. In fact, it MUST be a Boolean expression.

The while loop follows the "test then execute" pattern; it tests the value of i before printing it. The test is "i less than 5" so when i is equal to 5 the result of the test is false and i is not printed when its value is 5. We will return to a slightly different form of this code in our discussion on debugging loops. The use of loops can introduce some very subtle errors.

### The Loop Body

The loop body can be one of the following:

* An empty statement. You will see more about empty statements in the discussion of for loops below. A simple semicolon (“;”) denotes an empty statement.
* A single statement.
* A block of statements. This is the most common loop body. When you have a number of statements that need to be executed for each iteration of the loop, you put all of those statements in a block. The code is placed within a pair of curly braces "{ }" — this is the code that is looped or iterated. You do not need a pair of braces if you are iterating only one statement. However, we strongly advise you always to use a pair of braces to reduce the chance of errors — errors that are not found by the compiler and that are very difficult to locate and fix! An empty while statement should not have curly braces.

Here is the code sample discussed earlier in this section:

int i;   
i = 1;   
while (i < 5) {   
      out.println("<br>value of i is " + i);   
      i = i + 1;  
}

What happens if we leave out the i = i + 1; statement like this?

int i;   
i = 1;   
while (i < 5) {   
      out.println("<br>value of i is " + i);   
}

What would be the output for this new code? Think about this...

We set *i* equal to 1 and then make the test "*i* less than 5" which is true. We then print the value of *i*, which is 1. Since the code inside the loop is done, we go back to the test. Since *i* is still 1, it is less than 5, so the test is true and we execute the print statement printing 1. Since the code inside the loop is done, we go back to the test. Since *i* is still 1, it is less than 5, so the test is true and we execute the print statement print 1. And so on.

There seems to be a pattern here — an infinite pattern. This is an infinite loop. It will print the number 1 forever or until you forcibly terminate the program or turn off the computer.

The output would start like this:

value of i is 1

value of i is 1

value of i is 1

value of i is 1

...

... to infinity ...

Thus, one subtle error to watch out for is failing to adjust the variables that participate in the loop guard expression.

### While-Loop Example

Let us look at a servlet example that uses a while-loop. In this case, the user chooses how many images the servlet should display; we use this parameter in our loop guard to iterate through the correct number of image tags. To use this example, your form will need to supply an integer via an input named numImages. Alternatively, you could send a GET request to your iCarnegie Workbench using the URL

http://127.0.0.1:4615/servlet/WhileLoopServlet?numImages=5

|  |  |
| --- | --- |
| 1:  2:  3:  4:  5:  6:  7:  8:  9: 10: 11: 12: 13: 14: 15: 16: 17: 18: 19: 20: 21: 22: 23: 24: 25: 26: 27: 28: 29: 30: 31: 32: 33: 34: 35: 36: 37: 38: 39: 40: 41: 42: 43: 44: 45: 46: 47: 48: | import java.io.\*;  import javax.servlet.\*;  import javax.servlet.http.\*;  public class WhileLoopServlet extends HttpServlet {  private int numImages;  public void doGet(  HttpServletRequest request,  HttpServletResponse response)  throws ServletException, IOException {  doPost(request, response);  }  public void doPost(  HttpServletRequest request,  HttpServletResponse response)  throws ServletException, IOException {  String numImagesStr;  response.setContentType("text/html");  PrintWriter out = response.getWriter();  numImagesStr = request.getParameter("numImages");  numImages = Integer.parseInt(numImagesStr);  /\*\*  \* Write the response  \*/  out.println("<!DOCTYPE HTML PUBLIC '-//W3C//DTD "  + "HTML 4.01 Transitional//EN'>");  out.println("<html><head><title> Images - while loop "  + "</title></head><body>");  int i = 0;  while(i < numImages) {  out.println("<img src='/Animal.jpeg' alt='animal'>");  ++i;  }  out.println("</body></html>");  }  } |
| Listing 1 *[While-Loop Example](http://www.icarnegie.com/content/SSD/SSD1/2.1/normal/pg-intro-java-oop/pg-java-fund/pg-iteration/WhileLoopServlet.java" \t "external)* | |

As you will see in our next example, we can achieve the same effect in a more graceful way using a for-loop.

### The For-Loop

Another loop form in Java is the for-loop. It is true that you can do anything with a for-loop that you can do with a while-loop. While you can play a game of golf with only a putter, you can play a better game of golf with a bag of clubs. Similarly, you can write better programs if you use all three loops. The for-loop is best used when you know exactly how many iterations are expected.

It turns out that if you understand one loop and can use it correctly, it is straightforward to learn to use the other two.

|  |  |
| --- | --- |
| **Form of a for-loop** | **Diagrammatic representation of how the loop is executed by the computer.** |
| for (initialization; loopGuard; update)  loopBody |  |

Table 2 Form of a *for* loop

In the initialization part, you typically initialize the loop variable. In the following example, the loop variable i is initialized to 0. After initialization, the loopGuard is evaluated. As in the while loop, loopGuard is a Boolean expression. It should evaluate to true or false. If the loopGuard evaluates to false, the loop is terminated. After evaluating the loopGuard, and when it evaluates to true, the loopBody is executed. Then, the update is executed. In the following example, the update increments i by 1. After the update, the cycle of evaluating loopGuard - loopBody - update is repeated.

Here is a for-loop that does some counting:

int i;

for (i = 0; i < 5; i = i + 1){

out.println("value of i is " + i);

}

The output for this is:

value of i is 0

value of i is 1

value of i is 2

value of i is 3

value of i is 4

To examine the for-loop structure, let us write it in a slightly different format:

|  |  |
| --- | --- |
| **for** | the keyword "for" is part of Java — it tells Java that a loop is present |
| **(** | the opening parenthesis is required by Java |
| **i = 0;** | this is the initialization of the control variable, i — it happens first and only one time |
| **i < 5;** | this is the Boolean test — it happens second and is repeated one time for each subsequent iteration, and one more time at the end of the loop (if the loop iterates at least once). |
| **i = i+1;** | this is for the update of the control variable i (and is typically an increment or decrement) — it happens fourth and is repeated one time for each iteration |
| **)** | the closing parenthesis is required by Java |
| **{** | the opening brace begins the body of the loop — the next statement makes up the loop body; it is executed third and is repeated once for each iteration   * out.println("value of i is " + i); |
| **}** | the closing brace ends the body of the loop |

The pattern of execution for the for-loop is shown below

for(

***#1*** i = 0;

***#2*** i < 5;

***#4*** i = i + 1)

{

***#3*** out.println("value of i is " + i);

}

Let ***#5*** be the next instruction.

Then, the order of execution is:

#1-> #2 -> if #2 is true -> #3 -> #4 -> #2

if #2 is true -> #3 -> #4 -> #2

...

...

if #2 is true -> #3 -> #4 -> #2

if #2 is false -> #5

This continues until #2 is false. Once #2 is false, execution drops to the next instruction in the program.

A loop body can also contain an empty statement. As in the while-loop, the loop body can be an empty statement, a single statement, or a block of statements. We mentioned earlier that you might feel the need to "sleep" the program (i.e., slow down the execution of the program) while animating some drawing. It is possible to sleep with empty statements in the loop body of the for-loop:

for (i = 0; i < 1000; i = i+1);

Notice that the loop body does nothing because of the empty statement. Hence, the loop counts from 0 to 1000 and then the loop is done. An example of a for-loop with one statement is given below. Notice that the loopBody is a code block that contains a single if statement.

System.out.println ("Here are the even numbers between 1 and 10:");

for (i = 1; i < 10; i= i + 1) {

if (i % 2 == 0) {

out.println(i);  
    }  
}

### For-Loop Example

The functionality of this example is identical to our earlier [While-Loop Example](http://www.icarnegie.com/content/SSD/SSD1/2.1/normal/pg-intro-java-oop/pg-java-fund/pg-iteration/pg-iteration.html" \l "while-example#while-example). In this case, however, we use a for-loop to generate the image tags. Note that the use of the for-loop leads to a more natural program than our earlier example. Also, note that the for-loop helps avoid the subtle error of forgetting to perform an update of the control variable, since doing so is part of the statement's syntax.

|  |  |
| --- | --- |
| 1:  2:  3:  4:  5:  6:  7:  8:  9: 10: 11: 12: 13: 14: 15: 16: 17: 18: 19: 20: 21: 22: 23: 24: 25: 26: 27: 28: 29: 30: 31: 32: 33: 34: 35: 36: 37: 38: 39: 40: 41: 42: 43: 44: 45: 46: | import java.io.\*;  import javax.servlet.\*;  import javax.servlet.http.\*;  public class ForLoopServlet extends HttpServlet {  private int numImages;  public void doGet(  HttpServletRequest request,  HttpServletResponse response)  throws ServletException, IOException {  doPost(request, response);  }  public void doPost(  HttpServletRequest request,  HttpServletResponse response)  throws ServletException, IOException {  String numImagesStr;  response.setContentType("text/html");  PrintWriter out = response.getWriter();  numImagesStr = request.getParameter("numImages");  numImages = Integer.parseInt(numImagesStr);  out.println("<!DOCTYPE HTML PUBLIC '-//W3C//DTD "  + "HTML 4.01 Transitional//EN'>");  out.println("<html><head><title> Images - for loop "  + "</title></head><body>");  /\*  \* Write the response.  \* Note: we use increment operator (++i)  \* instead of i = i + 1  \*/  for (int i = 0; i < numImages; ++i) {  out.println("<img src='/Animal.jpeg' alt='animal'>");  }  out.println("</body></html>");  }  } |
| Listing 2 *[For-Loop Example](http://www.icarnegie.com/content/SSD/SSD1/2.1/normal/pg-intro-java-oop/pg-java-fund/pg-iteration/ForLoopServlet.java" \t "external)* | |

### Working with Iteration

While iteration is not a difficult concept in and of itself, mastering loops can be difficult without some additional tools. You will learn to use one of these tools, collections, in the next section of the course. In the meantime, review your specification for the simulation program and consider what pieces of the specification require iteration. Understanding loops also requires practice; the activity below will give you the opportunity to use loops in a practical way.

### Student Activity

We recommend that you complete the following activity. You will practice the following important skills:

* Using loops
* Using if-else-statements

We ask that you create a servlet-based photo gallery. You are provided PhotoGalleryServlet.class and a HTML Form as a sample solution. When you launch the HTML Form from the iCarnegie Servlet Workbench, you can enter the number of images to be displayed per page. The servlet returns an html page that contains the specified number of images, along with a "next" button to see the next set of images. This "next" button appears only if there are more images to be displayed.

Fifteen images (named 1.jpg, 2.jpg ...15.jpg) are provided to you. Create your own version of PhotoGalleryServlet that provides the same functionality as the sample. We recommend that you do this in two steps:

1. Create a servlet that displays the number of images entered by the user but does not provide a "next" button. The servlet should deal gracefully with a user entering a number that exceeds the number of images available. The for-loop and while-loop examples provided above should help you get started. Once you have this portion of the activity working, move on to the next step.
2. Implement the "next" functionality provided in the sample solution. In order to implement this functionality, you will need to pass two pieces of information to the servlet: the first image to be displayed on the page and the number of images that should appear on the page. This can be done using hidden form elements.

#### Files:

* [1.jpg](http://www.icarnegie.com/content/SSD/SSD1/2.1/normal/pg-intro-java-oop/pg-java-fund/pg-iteration/iteration-student-activity/1.jpg" \t "externalWindow) [2.jpg](http://www.icarnegie.com/content/SSD/SSD1/2.1/normal/pg-intro-java-oop/pg-java-fund/pg-iteration/iteration-student-activity/2.jpg" \t "externalWindow) [3.jpg](http://www.icarnegie.com/content/SSD/SSD1/2.1/normal/pg-intro-java-oop/pg-java-fund/pg-iteration/iteration-student-activity/3.jpg" \t "externalWindow) [4.jpg](http://www.icarnegie.com/content/SSD/SSD1/2.1/normal/pg-intro-java-oop/pg-java-fund/pg-iteration/iteration-student-activity/4.jpg" \t "externalWindow) [5.jpg](http://www.icarnegie.com/content/SSD/SSD1/2.1/normal/pg-intro-java-oop/pg-java-fund/pg-iteration/iteration-student-activity/5.jpg" \t "externalWindow) [6.jpg](http://www.icarnegie.com/content/SSD/SSD1/2.1/normal/pg-intro-java-oop/pg-java-fund/pg-iteration/iteration-student-activity/6.jpg" \t "externalWindow)[7.jpg](http://www.icarnegie.com/content/SSD/SSD1/2.1/normal/pg-intro-java-oop/pg-java-fund/pg-iteration/iteration-student-activity/7.jpg" \t "externalWindow) [8.jpg](http://www.icarnegie.com/content/SSD/SSD1/2.1/normal/pg-intro-java-oop/pg-java-fund/pg-iteration/iteration-student-activity/8.jpg" \t "externalWindow) [9.jpg](http://www.icarnegie.com/content/SSD/SSD1/2.1/normal/pg-intro-java-oop/pg-java-fund/pg-iteration/iteration-student-activity/9.jpg" \t "externalWindow) [10.jpg](http://www.icarnegie.com/content/SSD/SSD1/2.1/normal/pg-intro-java-oop/pg-java-fund/pg-iteration/iteration-student-activity/10.jpg" \t "externalWindow) [11.jpg](http://www.icarnegie.com/content/SSD/SSD1/2.1/normal/pg-intro-java-oop/pg-java-fund/pg-iteration/iteration-student-activity/11.jpg" \t "externalWindow) [12.jpg](http://www.icarnegie.com/content/SSD/SSD1/2.1/normal/pg-intro-java-oop/pg-java-fund/pg-iteration/iteration-student-activity/2.jpg" \t "externalWindow) [13.jpg](http://www.icarnegie.com/content/SSD/SSD1/2.1/normal/pg-intro-java-oop/pg-java-fund/pg-iteration/iteration-student-activity/13.jpg" \t "externalWindow) [14.jpg](http://www.icarnegie.com/content/SSD/SSD1/2.1/normal/pg-intro-java-oop/pg-java-fund/pg-iteration/iteration-student-activity/14.jpg" \t "externalWindow) [15.jpg](http://www.icarnegie.com/content/SSD/SSD1/2.1/normal/pg-intro-java-oop/pg-java-fund/pg-iteration/iteration-student-activity/5.jpg" \t "externalWindow)
* Alternatively, you may download all 15 images in a single [images.zip](http://www.icarnegie.com/content/SSD/SSD1/2.1/normal/pg-intro-java-oop/pg-java-fund/pg-iteration/iteration-student-activity/images.zip) archive.
* [PhotoGalleryForm.html](http://www.icarnegie.com/content/SSD/SSD1/2.1/normal/pg-intro-java-oop/pg-java-fund/pg-iteration/iteration-student-activity/PhotoGalleryForm.html" \t "externalWindow)
* [PhotoGalleryServlet.class](http://www.icarnegie.com/content/SSD/SSD1/2.1/normal/pg-intro-java-oop/pg-java-fund/pg-iteration/iteration-student-activity/PhotoGalleryServlet.class)

**2.3.6 Using Vector**

Recall from [2.2.3 Lifecycle of Objects](javascript:ContentByName('pg-objects-lifecycle');) that a Vector object acts like a container of object references. We discussed how to create a Vector object and how to add object references to it. Recall that a Vector contains only references to objects, not objects themselves. In this page, we will use the context of a Catfish class to perform the following tasks with a Vector:

* Getting a specific object reference from a Vector
* Getting the size of a Vector, i.e., the number of references stored in a Vector

**Adding New Living Beings to Simulation**

Let us first examine how catfish, algae, and crocodile objects are added to the simulation. By clicking various checkboxes in the HTML form, the user specifies the organisms to create and their locations. When the user clicks the "Start Simulation" button, the SimulationServlet receives the browser POST request. The SimulationServlet invokes the createLivingBeing method of the LivingBeing class for each living organism to be created.

|  |  |
| --- | --- |
| 1:  2:  3:  4:  5:  6:  7:  8:  9: 10: 11: 12: 13: 14: 15: 16: 17: 18: 19: 20: 21: 22: 23: 24: 25: 26: 27: 28: 29: 30: 31: | /\*\*  \* Create an organism that belongs to a specified species at  \* a specified location and add it to the specified simulation.  \*  \* @param simulation - the simulation this organism belongs to  \* @param species - the organism's species  \* @param value - row and column values specified as String from HTML form.  \* row value is the last two digits,  \* column value is given in the remaining digits.  \*/  public static void createLivingBeing(  Simulation sim,  String species,  String value) {  int rowAndCol = Integer.parseInt(value);  int row = rowAndCol / 100;  int column = rowAndCol - (100 \* row);  if (species.equals("algae")) {  sim.addLivingBeing(new AlgaeColony(row, column, sim));  }  if (species.equals("catfish")) {  sim.addLivingBeing(new Catfish(row, column, sim));  }  if (species.equals("crocodile")) {  sim.addLivingBeing(new Crocodile(row, column, sim));  }  } |
| Listing 1 *createLivingBeing* | |

Based on the species value, the createLivingBeing method creates a new algae or catfish or crocodile object at the specified location. In addition, it adds the new object to the simulation by sending an addLivingBeing message with the reference to the newly created object. If you decide to add a new species to the Simulation program, you need to add functionality to the createLivingBeing method to accommodate the new species. In addition, you must implement that new species as a class that extends LivingBeing.

After all living being objects are added to the Simulation object, the Simulation object sends a liveALittle message to each living being object. Our catfish chooses to live by eating and swimming. When a Catfish object gets its turn to live, it should look for food (algae) if hungry and eat food if available. Let us examine the code needed to look for food.

**Catfish Looks for Food**

Let us first state in high-level English how we would implement "looking for food." Then we will convert our English statements into Java statements:

* Spend ENERGY\_TO\_LOOK\_FOR\_FOOD
* If I am dead, return immediately.
* Ask simulation object to give the list of all my neighbors.
* If a neighbor is algae, then that neighbor is food. Return that neighbor.
* If none of the neighbors is algae, return null.

We translate the English statements into Java as follows. Some things may be new to you that we will explain shortly.

|  |  |
| --- | --- |
| 1:  2:  3:  4:  5:  6:  7:  8:  9: 10: 11: 12: 13: 14: 15: 16: 17: 18: 19: 20: 21: 22: 23: 24: 25: 26: 27: 28: 29: 30: | /\*\*  \* Look for food in the neighborhood. Consume some energy in the process.  \*  \* @return a neighboring algae that is food.  \*/  private AlgaeColony lookForFoodInNeighborhood() {  int neighborIndex;  // Looking for food consumes energy.  setEnergy(getEnergy() - ENERGY\_TO\_LOOK\_FOR\_FOOD);  if (isDead()) {  return null;  }  Vector neighbors =  simulation.getNeighbors(getRow(), getColumn(), 1);  for (neighborIndex = 0;  neighborIndex < neighbors.size();  ++neighborIndex) {  if (neighbors.get(neighborIndex) instanceof AlgaeColony) {  return (AlgaeColony) neighbors.get(neighborIndex);  }  }  return null;  } |
| Listing 2 *lookForFoodInNeighborhood* | |

We ask the simulation object to give us all the living beings that are in the catfish's neighborhood. Neighborhood for a catfish is that portion of the lake that is one cell away from the fish, and hence the parameter value of 1 is passed to getNeighbors.

**Iterating through Elements in a Vector**

The getNeighbors method returns a reference to a Vector object. This is the first time we are using a method that returns a Vector reference. We store that reference in a local variable named neighbors.

In English, we can easily state, "Select the subset of living beings that are algae from this set of living beings that are neighbors." Unfortunately, the Java language does not have any construct to perform such an operation. We have to inspect each member of the set to do our selection. We use a for-loop to do that. The loop iterates through each member of the set starting from the element at index 0 and going up to (and including) the element at index size - 1. Notice the use of Vector's size method. Notice also the use of the increment operator ++, which we discussed in [2.3.2 Arithmetic Operators and Expressions](javascript:ContentByNameAnchor('pg-arithmetic-operators');).

**instanceof Operator**

We know that each object has to be an instance of some class. For example, a catfish object is an instance of the Catfish class. In OOP, it is possible for an object to be an instance of multiple classes at the same time. All algae objects are also instances of the LivingBeing class because AlgaeColony.java has declared that AlgaeColony extends LivingBeing. When AlgaeColony extends LivingBeing, AlgaeColony is said to "inherit" from LivingBeing. We will get into inheritance in [Unit 3 Inheritance](javascript:ContentByName('pg-inheritance');). For now, all we need to know is that, in Java, all classes implicitly extend the class named [Object](http://java.sun.com/j2se/1.4.2/docs/api/java/lang/Object.html" \t "externalWindow). Hence, all objects are instances of the class Object. A simpler approach is to view each AlgaeColony object as an instance of AlgaeColony class *or* as an instance of the Object class. We can take either view to suit our convenience.

In Java, a Vector stores references to Object instances. So, although we know that objects in our neighbors Vector are instances of LivingBeing (because the documentation for Simulation says so), the Vector class knows these objects only as instances of Object class. Thus, when we send a get message to the Vector object, it returns a reference to an Object. Since that reference may refer to an instance of Catfish, AlgaeColony, or Crocodile, we have to inspect its type to see if it is algae. We use the instanceof operator to inspect its type. Recall from [2.3.3 Boolean and Relational Operators and Expressions](javascript:ContentByName('pg-boolean-and-relational-operators');) that the instanceof operator is a relational operator that compares two entities. The two entities are a *reference* and a *reference type* (i.e., a class or null). The result of the comparison is true only if the *reference* refers to an instance of the specified *type*.

**Casting**

The get method returns a reference of type Object. If the reference refers to an instance of AlgaeColony, we need to cast the reference type from Object to AlgaeColony. You can cast a value of some type to another type by specifying the destination type in parentheses:

(AlgaeColony) neighbors.get(neighborIndex)

Note: If your code performs a cast and the object is not really the type to which it is cast, a run-time error occurs.

Returning to our example, if we do not find any algae neighbor, there is no food in our neighborhood, and we return null. That concludes our search for food.

In the context of the bank teller example, please look at the implementation of the lookUpAccount method of AccountsLedger class for an example of a for-loop, a Vector, and casting.

**Student Activity**

We recommend that you complete the following activity. You will practice the following important skills:

* The ability to create a collection of objects (using Vector)
* The ability to iterate through elements of a Vector

The class below, AnimalGalleryServlet, has functionality similar to that of your earlier PhotoGalleryServlet. Users enter the number of images to be displayed per page. In response, the page displays the specified number of images and some information about each image.

Modify your PhotoGalleryServlet.java to create an AnimalGalleryServlet.java. Start by creating a Vector of Animal objects (Animal.java is provided below). For each animal reference in your vector, use the Animal's accessors (get methods) to get the appropriate information. Use Vector's size method to determine the maximum number of images to be displayed. You can reuse your images from the student activity in [2.3.5 Iteration](javascript:ContentByName('pg-iteration');). The text file, image\_info.txt gives information about each image.

You will need the following files to complete this activity:

* [AnimalGalleryForm.html](http://www.icarnegie.com/content/SSD/SSD1/2.1/normal/pg-intro-java-oop/pg-java-fund/pg-vector/vectors-student-activities/AnimalGalleryForm.html" \t "externalWindow)
* [Animal.java](http://www.icarnegie.com/content/SSD/SSD1/2.1/normal/pg-intro-java-oop/pg-java-fund/pg-vector/vectors-student-activities/Animal.java" \t "externalWindow)
* [AnimalGalleryServlet.class](http://www.icarnegie.com/content/SSD/SSD1/2.1/normal/pg-intro-java-oop/pg-java-fund/pg-vector/vectors-student-activities/AnimalGalleryServlet.class)
* [image\_info.txt](http://www.icarnegie.com/content/SSD/SSD1/2.1/normal/pg-intro-java-oop/pg-java-fund/pg-vector/vectors-student-activities/image_info.txt" \t "externalWindow)

**Unit 3. Inheritance**

In [2.3.6 Using Vector](javascript:ContentByName('pg-vector');), you learned that, because of inheritance, it is possible for an object to be an instance of more than one class. You have already seen this in the life simulation. The AlgaeColony class extended the LivingBeing class. We will now examine inheritance in greater detail. We will show you how you can use inheritance to organize your classes in an intuitive manner.

**3.1 Introduction to Inheritance**

**Introduction**

Biological inheritance has a great impact on the way we are. We have inherited some of our characteristics from our parents. Our parents inherited some of those characteristics from their parents. The concept of inheritance can also be applied to categorical descriptions of objects. When applying inheritance to objects, objects may be classified in a top-down fashion from the least specific to the most specific form of an object.

Consider bank accounts. All bank accounts share common characteristics such as a balance and a means of identification, usually in the form of a name or account number. Bank accounts can be further subdivided into groups such as savings accounts, and checking accounts. If we were to code Java classes to represent checking accounts and savings accounts, we would define much of the same code for both classes. They would both have a name and a balance as well as methods to deposit and withdraw money. It would be much more efficient to define the common behavior of the two account types in one place. We would then only have to deal with the differences between the account types when we write the classes.

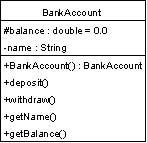
Object-oriented programming languages allow programmers to design objects in a top-down fashion using inheritance. Object-oriented programming languages gives the programmer the ability to define a general form of an object and then use inheritance to create more specific forms of the object. Using inheritance, an object acquires all the methods and attributes of the class from which it inherits. This means that class has the attributes and behavior specified in parent classes without explicitly declaring them. Those methods and attributes are in addition to the methods and attributes declared directly in a class.

**Terminology**

When categorizing objects, one may do so from the least to the most specific, or from the most to the least specific. The process of recognizing the common characteristics of existing objects and collapsing them into a more general form is known as *generalization*. The opposite of generalization, deriving specific forms of objects, is known as *specialization*. The relationship that is shared between specific and general forms of objects is known as the "is-a" relationship. An object of a specific form "is-a" object of the general form as well. For example, an animal *is a* type of living being. There are a number of synonymous names for the class that other classes inherit from. Some of these synonymous names are *super class*, *parent class*, *base class*, or *supertype*. The new class that inherits from the parent class is known as the *subclass*, *derived class*, or *subtype*.

**Inheritance in Action**

It may seem very abstract at this point that a class can use the attributes and methods of another class without declaring them. To help you visualize inheritance, we will now build a number of classes based on bank accounts. Bank accounts, implemented in the BankAccount class, have instance variables balance and name as well as methods that get the name and balance of an account, getName, and getBalance respectively. The figure below details the BankAccount class and its instance variables:

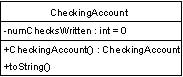
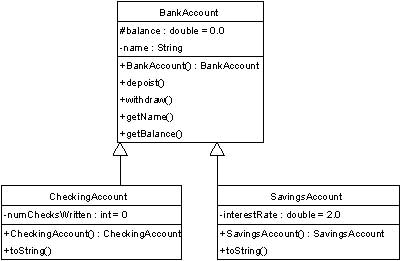


The code for the BankAccount class follows:

|  |  |
| --- | --- |
| 1:  2:  3:  4:  5:  6:  7:  8:  9: 10: 11: 12: 13: 14: 15: 16: 17: 18: 19: 20: 21: 22: 23: 24: 25: 26: 27: 28: 29: 30: 31: 32: 33: 34: 35: 36: 37: 38: 39: 40: 41: 42: 43: 44: 45: 46: 47: 48: 49: 50: 51: 52: 53: 54: 55: 56: 57: 58: 59: 60: 61: 62: 63: 64: 65: | /\*\*  \* Class modeling a simple BankAccount.  \*  \* @author iCarnegie vp  \*/  public class BankAccount {  /\*\*  \* BankAccount's balance  \*/  protected double balance = 0.0;  /\*\*  \* BankAccount's name  \*/  private String name;  /\*\*  \* BankAccount's constructor takes initialName and initialBalance  \* as input.  \*  \* @param initialName - name of the account  \* @param initialBalance - balance of the account  \*/  public BankAccount(String initialName, double initialBalance) {  name = initialName;  balance = initialBalance;  }  /\*\*  \* Deposit money into the account.  \*  \* @param amount - amount of money to deposit.  \*/  public void deposit(int amount) {  balance = balance + amount;  }  /\*\*  \* Withdraw money from an account.  \*  \* @param amount - amount of money to withdraw.  \*/  public void withdraw(int amount) {  balance = balance - amount;  }  /\*\*  \* Return the name of the account.  \*  \* @return - a string representing the account name.  \*/  public String getName() {  return name;  }  /\*\*  \* Return the balance of the account.  \*  \* @return - a double representing the account balance.  \*/  public double getBalance() {  return balance;  }  } |
| **Listing 1***[The BankAccount class](http://www.icarnegie.com/content/SSD/SSD1/2.1/normal/pg-inheritance/pg-introtoinheritance/BankAccount.java" \t "external)* | |

You may have noticed an unfamiliar access modifier in the BankAccount class. We will discuss the protected keyword in the next page.

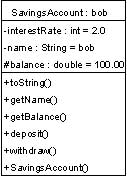
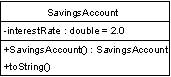
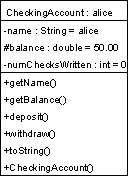
We will create two subclasses of the BankAccount class. These classes will represent checking accounts and savings accounts. Checking accounts are implemented in the CheckingAccount class. Savings accounts are implemented in the SavingsAccount class. The figure below shows the hierarchy of the BankAccount, CheckingAccount, and SavingsAccount classes.



The CheckingAccount class contains one instance variable, numChecksWritten that represents the number of checks that have been written against an account. The CheckingAccount class also has a name and a balance that it received from the BankAccount class via inheritance. The figure below is a pictorial representation of the CheckingAccount class.

Notice from the above picture that instance variables and methods of BankAccount are not repeated in CheckingAccount.

Recall that the extends keyword tells Java that the class being declared inherits from the class whose name immediately follows the extends keyword. In this case, CheckingAccount extends BankAccount; therefore, it is a subclass of BankAccount. Notice that the CheckingAccount class contains only one instance variable, numChecksWritten, and one method, a constructor. Suppose we were to create an instance of the CheckingAccount class named alice. This instance is used to represent the checking account owned by *alice* with balance $50.00. Note the figure below. Even though the CheckingAccount class declares one instance variable, an instance of it will also have a name and a balance.



The SavingsAccount class differs from the CheckingAccount class in that, in addition to the instance variables and methods that it inherits from BankAccount, it also stores the interest rate that the savings account earns. The interest rate is stored in the interestRate instance variable. Consider the figure below detailing the SavingsAccount class.

Let us consider an instance of the SavingsAccount class that stored information about a savings account owned by *bob* with balance $100.00 and earning a 2% interest rate. Note from the figure below that even though the SavingsAccount class does not declare a name and a balance, it inherits them from the BankAccount class.

**Advantages of Programming with Inheritance**

There are a number of advantages to using inheritance when designing software systems. First, inheritance allows programmers to design systems in an intuitive, top-down fashion. Second, it provides a framework for code re-use.

Humans naturally prefer classification when organizing information. Consider a library. Books in a library are divided into categories to make them easier for humans to search. All fiction books are grouped together and all non-fiction books are grouped together. Non-fiction books are further subdivided into such categories as histories and biographies. Fiction books are subdivided among such categories as science fiction, mysteries, and techno-thrillers. Programmers can design complex systems in a top-down fashion from the least specific class to most specific class using inheritance. Inheritance lets you define shared behavior at a high-level and all objects that inherit from the original class acquire that behavior. When the behavior of an object is faulty and requires fixing a logic error in the code, that fix need only be made in one place and all subclasses benefit.

Let us explore the use of shared behavior in the context of the fish example. By now, you should have created classes that represent algae colonies, crocodiles, and catfish. Recall that the Catfish, AlgaeColony, and Crocodile classes all had a function named die that operated the same way for all classes. Consider what would happen if there was an error in your die method that needed to be corrected. Changing the behavior of the die method means that you must make the change in three classes. If you used inheritance, you could make necessary changes to shared behavior at a higher level. All classes that inherit from it can make use of that new behavior. When that behavior needs to change, you need to make only one change that affects all classes.

Inheritance also provides a framework for code re-use to programmers. This concept is not completely orthogonal to designing systems in a top-down fashion. While you were coding Catfish, AlgaeColony, and Crocodile classes, you may have found yourself generating a lot of the same code for each class — attributes and methods had the same name and functionality in many of the classes. For example, Crocodile, Catfish, and AlgaeColony each had attributes named age, energy and isAlive, and had methods named getAge() and getEnergy(). This can be implemented more efficiently using inheritance.

We can define another class, in addition to AlgaeColony, Crocodile, and Catfish, from which all the aforementioned classes may derive common attributes and methods. Previously, you were given an implementation of the class named LivingBeing. Each of the classes you wrote extended LivingBeing, which contained no common attributes and behavior. We will provide you with a more detailed implementation of the LivingBeing class where the common attributes and behaviors of the AlgaeColony, Catfish, and Crocodile classes are implemented. Thus, LivingBeing is the most general form of the AlgaeColony, Crocodile, and Catfish objects in the life simulation. Indeed, they are all living beings. The LivingBeing class serves as a super class from which the subclasses AlgaeColony, Crocodile, and Catfish derive attributes and methods. We can define attributes and methods common to AlgaeColony, Crocodile, and Catfish in the LivingBeing class. All classes that inherit from LivingBeing may refer to these attributes and methods without declaring them themselves. The living beings in the fish simulation may be further categorized between those that are animals and those that are not. While AlgaeColony, Crocodile, and Catfish may all be living beings, Crocodile and Catfish share functionality that AlgaeColony does not. Crocodile and Catfish can, for example, move and eat. The ability to move and eat is shared by all animals, and, hence, we will create a new class named Animal. The common functionality that Crocodile and Catfish share can be subsumed into a class known as Animal, which is a subclass of LivingBeing and a superclass of Catfish and Crocodile. Creating this new superclass will also allow use to create classes for different animals in the future, frogs for example.

**When Not to Use Inheritance**

Now, it may seem as though all common attributes and methods should be taken from specialized forms of objects and placed into superclasses. But, some attributes and methods do not belong in a super class. Indeed, adding some of these attributes and methods may lead to semantically incorrect behavior. There are some attributes and methods that do not belong to all living beings and should not be part of the living being class. All living beings have an age, an energy level and are alive (or not). This common functionality should be generalized into the LivingBeing class. However, consider the ability to move. It does not make sense to include the ability to move in the LivingBeing class since AlgaeColony cannot move. The ability to move, however, should be part of the Animal class because all animals can move. Only those methods and attributes that belong to a superclass should be generalized into a superclass.

**Inheritance in Java**

The Java standard API was built using inheritance. Navigate a web browser to [The Java Version 1.4.2 API Specification](http://java.sun.com/j2se/1.4.2/docs/api/index.html" \t "anewwindow) that details all classes, including their methods and attributes, that are available as part of the standard Java language. In the left frame, named *All Classes*, search for the class named String, and click on the link for the String class. Near the top of the page are the words public final class String extends Object. This states that the String class is a subclass of Object. In Java, all classes are subclasses of Object and inherit all of Object's attributes and methods. Search on the same page for the text *Methods inherited from class java.lang.Object*. The methods listed are those that String has inherited from the Object class. For example, the Object class defines a method named toString that returns a String representation of an object. This is typically used to build output statements that are sent to the system console.

In conclusion, we have seen how inheritance may be used to design objects in a manner that is natural for humans. We can design objects starting with their least specific forms, where all general behavior is shared in one place. We can then proceed to derive specific versions of these general objects. We have also seen that designing software in this manner provides a framework for code re-use.

**3.1 Introduction to Inheritance**

**Introduction**

Biological inheritance has a great impact on the way we are. We have inherited some of our characteristics from our parents. Our parents inherited some of those characteristics from their parents. The concept of inheritance can also be applied to categorical descriptions of objects. When applying inheritance to objects, objects may be classified in a top-down fashion from the least specific to the most specific form of an object.

Consider bank accounts. All bank accounts share common characteristics such as a balance and a means of identification, usually in the form of a name or account number. Bank accounts can be further subdivided into groups such as savings accounts, and checking accounts. If we were to code Java classes to represent checking accounts and savings accounts, we would define much of the same code for both classes. They would both have a name and a balance as well as methods to deposit and withdraw money. It would be much more efficient to define the common behavior of the two account types in one place. We would then only have to deal with the differences between the account types when we write the classes.

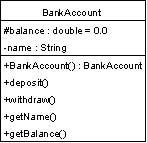
Object-oriented programming languages allow programmers to design objects in a top-down fashion using inheritance. Object-oriented programming languages gives the programmer the ability to define a general form of an object and then use inheritance to create more specific forms of the object. Using inheritance, an object acquires all the methods and attributes of the class from which it inherits. This means that class has the attributes and behavior specified in parent classes without explicitly declaring them. Those methods and attributes are in addition to the methods and attributes declared directly in a class.

**Terminology**

When categorizing objects, one may do so from the least to the most specific, or from the most to the least specific. The process of recognizing the common characteristics of existing objects and collapsing them into a more general form is known as *generalization*. The opposite of generalization, deriving specific forms of objects, is known as *specialization*. The relationship that is shared between specific and general forms of objects is known as the "is-a" relationship. An object of a specific form "is-a" object of the general form as well. For example, an animal *is a* type of living being. There are a number of synonymous names for the class that other classes inherit from. Some of these synonymous names are *super class*, *parent class*, *base class*, or *supertype*. The new class that inherits from the parent class is known as the *subclass*, *derived class*, or *subtype*.

**Inheritance in Action**

It may seem very abstract at this point that a class can use the attributes and methods of another class without declaring them. To help you visualize inheritance, we will now build a number of classes based on bank accounts. Bank accounts, implemented in the BankAccount class, have instance variables balance and name as well as methods that get the name and balance of an account, getName, and getBalance respectively. The figure below details the BankAccount class and its instance variables:

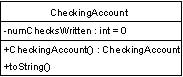
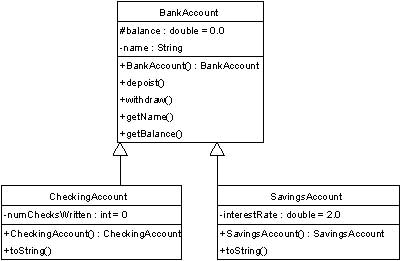


The code for the BankAccount class follows:

|  |  |
| --- | --- |
| 1:  2:  3:  4:  5:  6:  7:  8:  9: 10: 11: 12: 13: 14: 15: 16: 17: 18: 19: 20: 21: 22: 23: 24: 25: 26: 27: 28: 29: 30: 31: 32: 33: 34: 35: 36: 37: 38: 39: 40: 41: 42: 43: 44: 45: 46: 47: 48: 49: 50: 51: 52: 53: 54: 55: 56: 57: 58: 59: 60: 61: 62: 63: 64: 65: | /\*\*  \* Class modeling a simple BankAccount.  \*  \* @author iCarnegie vp  \*/  public class BankAccount {  /\*\*  \* BankAccount's balance  \*/  protected double balance = 0.0;  /\*\*  \* BankAccount's name  \*/  private String name;  /\*\*  \* BankAccount's constructor takes initialName and initialBalance  \* as input.  \*  \* @param initialName - name of the account  \* @param initialBalance - balance of the account  \*/  public BankAccount(String initialName, double initialBalance) {  name = initialName;  balance = initialBalance;  }  /\*\*  \* Deposit money into the account.  \*  \* @param amount - amount of money to deposit.  \*/  public void deposit(int amount) {  balance = balance + amount;  }  /\*\*  \* Withdraw money from an account.  \*  \* @param amount - amount of money to withdraw.  \*/  public void withdraw(int amount) {  balance = balance - amount;  }  /\*\*  \* Return the name of the account.  \*  \* @return - a string representing the account name.  \*/  public String getName() {  return name;  }  /\*\*  \* Return the balance of the account.  \*  \* @return - a double representing the account balance.  \*/  public double getBalance() {  return balance;  }  } |
| **Listing 1***[The BankAccount class](http://www.icarnegie.com/content/SSD/SSD1/2.1/normal/pg-inheritance/pg-introtoinheritance/BankAccount.java" \t "external)* | |

You may have noticed an unfamiliar access modifier in the BankAccount class. We will discuss the protected keyword in the next page.

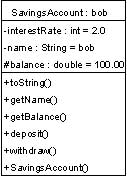
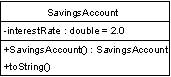
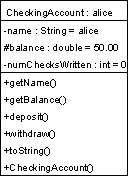
We will create two subclasses of the BankAccount class. These classes will represent checking accounts and savings accounts. Checking accounts are implemented in the CheckingAccount class. Savings accounts are implemented in the SavingsAccount class. The figure below shows the hierarchy of the BankAccount, CheckingAccount, and SavingsAccount classes.



The CheckingAccount class contains one instance variable, numChecksWritten that represents the number of checks that have been written against an account. The CheckingAccount class also has a name and a balance that it received from the BankAccount class via inheritance. The figure below is a pictorial representation of the CheckingAccount class.

Notice from the above picture that instance variables and methods of BankAccount are not repeated in CheckingAccount.

Recall that the extends keyword tells Java that the class being declared inherits from the class whose name immediately follows the extends keyword. In this case, CheckingAccount extends BankAccount; therefore, it is a subclass of BankAccount. Notice that the CheckingAccount class contains only one instance variable, numChecksWritten, and one method, a constructor. Suppose we were to create an instance of the CheckingAccount class named alice. This instance is used to represent the checking account owned by *alice* with balance $50.00. Note the figure below. Even though the CheckingAccount class declares one instance variable, an instance of it will also have a name and a balance.



The SavingsAccount class differs from the CheckingAccount class in that, in addition to the instance variables and methods that it inherits from BankAccount, it also stores the interest rate that the savings account earns. The interest rate is stored in the interestRate instance variable. Consider the figure below detailing the SavingsAccount class.

Let us consider an instance of the SavingsAccount class that stored information about a savings account owned by *bob* with balance $100.00 and earning a 2% interest rate. Note from the figure below that even though the SavingsAccount class does not declare a name and a balance, it inherits them from the BankAccount class.

**Advantages of Programming with Inheritance**

There are a number of advantages to using inheritance when designing software systems. First, inheritance allows programmers to design systems in an intuitive, top-down fashion. Second, it provides a framework for code re-use.

Humans naturally prefer classification when organizing information. Consider a library. Books in a library are divided into categories to make them easier for humans to search. All fiction books are grouped together and all non-fiction books are grouped together. Non-fiction books are further subdivided into such categories as histories and biographies. Fiction books are subdivided among such categories as science fiction, mysteries, and techno-thrillers. Programmers can design complex systems in a top-down fashion from the least specific class to most specific class using inheritance. Inheritance lets you define shared behavior at a high-level and all objects that inherit from the original class acquire that behavior. When the behavior of an object is faulty and requires fixing a logic error in the code, that fix need only be made in one place and all subclasses benefit.

Let us explore the use of shared behavior in the context of the fish example. By now, you should have created classes that represent algae colonies, crocodiles, and catfish. Recall that the Catfish, AlgaeColony, and Crocodile classes all had a function named die that operated the same way for all classes. Consider what would happen if there was an error in your die method that needed to be corrected. Changing the behavior of the die method means that you must make the change in three classes. If you used inheritance, you could make necessary changes to shared behavior at a higher level. All classes that inherit from it can make use of that new behavior. When that behavior needs to change, you need to make only one change that affects all classes.

Inheritance also provides a framework for code re-use to programmers. This concept is not completely orthogonal to designing systems in a top-down fashion. While you were coding Catfish, AlgaeColony, and Crocodile classes, you may have found yourself generating a lot of the same code for each class — attributes and methods had the same name and functionality in many of the classes. For example, Crocodile, Catfish, and AlgaeColony each had attributes named age, energy and isAlive, and had methods named getAge() and getEnergy(). This can be implemented more efficiently using inheritance.

We can define another class, in addition to AlgaeColony, Crocodile, and Catfish, from which all the aforementioned classes may derive common attributes and methods. Previously, you were given an implementation of the class named LivingBeing. Each of the classes you wrote extended LivingBeing, which contained no common attributes and behavior. We will provide you with a more detailed implementation of the LivingBeing class where the common attributes and behaviors of the AlgaeColony, Catfish, and Crocodile classes are implemented. Thus, LivingBeing is the most general form of the AlgaeColony, Crocodile, and Catfish objects in the life simulation. Indeed, they are all living beings. The LivingBeing class serves as a super class from which the subclasses AlgaeColony, Crocodile, and Catfish derive attributes and methods. We can define attributes and methods common to AlgaeColony, Crocodile, and Catfish in the LivingBeing class. All classes that inherit from LivingBeing may refer to these attributes and methods without declaring them themselves. The living beings in the fish simulation may be further categorized between those that are animals and those that are not. While AlgaeColony, Crocodile, and Catfish may all be living beings, Crocodile and Catfish share functionality that AlgaeColony does not. Crocodile and Catfish can, for example, move and eat. The ability to move and eat is shared by all animals, and, hence, we will create a new class named Animal. The common functionality that Crocodile and Catfish share can be subsumed into a class known as Animal, which is a subclass of LivingBeing and a superclass of Catfish and Crocodile. Creating this new superclass will also allow use to create classes for different animals in the future, frogs for example.

**When Not to Use Inheritance**

Now, it may seem as though all common attributes and methods should be taken from specialized forms of objects and placed into superclasses. But, some attributes and methods do not belong in a super class. Indeed, adding some of these attributes and methods may lead to semantically incorrect behavior. There are some attributes and methods that do not belong to all living beings and should not be part of the living being class. All living beings have an age, an energy level and are alive (or not). This common functionality should be generalized into the LivingBeing class. However, consider the ability to move. It does not make sense to include the ability to move in the LivingBeing class since AlgaeColony cannot move. The ability to move, however, should be part of the Animal class because all animals can move. Only those methods and attributes that belong to a superclass should be generalized into a superclass.

**Inheritance in Java**

The Java standard API was built using inheritance. Navigate a web browser to [The Java Version 1.4.2 API Specification](http://java.sun.com/j2se/1.4.2/docs/api/index.html" \t "anewwindow) that details all classes, including their methods and attributes, that are available as part of the standard Java language. In the left frame, named *All Classes*, search for the class named String, and click on the link for the String class. Near the top of the page are the words public final class String extends Object. This states that the String class is a subclass of Object. In Java, all classes are subclasses of Object and inherit all of Object's attributes and methods. Search on the same page for the text *Methods inherited from class java.lang.Object*. The methods listed are those that String has inherited from the Object class. For example, the Object class defines a method named toString that returns a String representation of an object. This is typically used to build output statements that are sent to the system console.

In conclusion, we have seen how inheritance may be used to design objects in a manner that is natural for humans. We can design objects starting with their least specific forms, where all general behavior is shared in one place. We can then proceed to derive specific versions of these general objects. We have also seen that designing software in this manner provides a framework for code re-use.

**3.2 Using Inheritance**

**Introduction**

In [3.1 Introduction to Inheritance](javascript:ContentByName('pg-introtoinheritance');), we introduced you to the concept of classes acquiring the attributes and methods declared in its superclasses via inheritance. Let us now take a close look at the nuances of using inheritance when designing classes. We will motivate our discussion by returning to the bank account example. In [3.1 Introduction to Inheritance](javascript:ContentByName('pg-introtoinheritance');), we created two subclasses of the BankAccount class, namely CheckingAccount and SavingsAccount. We will augment the functionality of these accounts as we explore inheritance in greater depth.

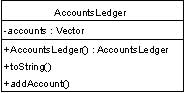
**Controlling Access**

A programmer needs to be aware of the effects that access control modifiers have when using inheritance. Access control modifiers determine what classes may access another class' methods and variables. Recall that variables and methods may have different access controls associated with them. Java provides three levels of access: public, private, and protected. You should already be familiar with public and private access. public access allows code from outside a class definition to access a variable or a method directly. private access does not allow access from outside a class. protected access allows access to methods and instance variables from within the class that defines them, or from all of its subclasses, or from all classes within the same package. Since a discussion of packages is not in the scope of SSD1, you should use protected access only for instance variables that you want to be visible to subclasses. Note the implementation of the BankAccount class below. The balance is a protected double. It is protected because we will want BankAccount's subclasses to access it and modify it. In particular, the deposit and withdraw methods will need to add to and subtract from balance. The instance variable name is left private because subclasses will not need to change an account's name.

|  |  |
| --- | --- |
| 1:  2:  3:  4:  5:  6:  7:  8:  9: 10: 11: 12: 13: 14: 15: 16: 17: 18: 19: 20: 21: 22: 23: 24: 25: 26: 27: 28: 29: 30: 31: 32: 33: 34: 35: 36: 37: 38: 39: 40: 41: 42: 43: 44: 45: 46: 47: 48: 49: 50: 51: 52: 53: 54: 55: 56: 57: 58: 59: 60: 61: 62: 63: 64: 65: | /\*\*  \* Class modeling a simple BankAccount.  \*  \* @author iCarnegie vp  \*/  public class BankAccount {  /\*\*  \* BankAccount's balance  \*/  protected double balance = 0.0;  /\*\*  \* BankAccount's name  \*/  private String name;  /\*\*  \* BankAccount's constructor takes initialName and initialBalance  \* as input.  \*  \* @param initialName - name of the account  \* @param initialBalance - balance of the account  \*/  public BankAccount(String initialName, double initialBalance) {  name = initialName;  balance = initialBalance;  }  /\*\*  \* Deposit money into the account.  \*  \* @param amount - amount of money to deposit.  \*/  public void deposit(int amount) {  balance = balance + amount;  }  /\*\*  \* Withdraw money from an account.  \*  \* @param amount - amount of money to withdraw.  \*/  public void withdraw(int amount) {  balance = balance - amount;  }  /\*\*  \* Return the name of the account.  \*  \* @return - a string representing the account name.  \*/  public String getName() {  return name;  }  /\*\*  \* Return the balance of the account.  \*  \* @return - a double representing the account balance.  \*/  public double getBalance() {  return balance;  }  } |
| **Listing 1** *[The BankAccount class](http://www.icarnegie.com/content/SSD/SSD1/2.1/normal/pg-inheritance/pg-usinginheritance/BankAccount.java" \t "external)* | |

**Polymorphism**

We may want two or more classes to respond to the same method call, or message, in their own ways. The term *polymorphism* refers to this ability. Consider a Vector of CheckingAccount and SavingsAccount objects. This Vector of accounts is encapsulated in the AccountsLedger class. The figure below depicts the AccountsLedger class.



Consider printing a list of the accounts in the ledger. We would like the toString method of the AccountsLedger to return a String that contains information about all of the accounts including the type of account (checking or savings), the name of the account, the balance, and the interest rate that the account earns if it is a savings account. One way to go about implementing this method is to have special code that tests the type of account and builds a string representation for each type. This is illustrated by the following implementation of the toString method.

|  |  |
| --- | --- |
| 1:  2:  3:  4:  5:  6:  7:  8:  9: 10: 11: 12: 13: 14: 15: 16: 17: 18: 19: | public String toString() {  String accountString = "";  // For each account in the vector, append the String representation  // of it to the String named accountString.  for (int i=0; i < accounts.size(); i++ ) {  if (accounts.get(i) instanceof CheckingAccount) {  accountString = accountString + "Checking account: "  + getName() + " balance: $ " + balance ;  }  if (accounts.get(i) instanceof SavingsAccount) {  accountString = accountString + "Savings account : "  + getName() + " balance: $ " + balance  + " earning " + interestRate  + " % interest";  }  }  return accountString;  } |
| **Listing 2** *A possible toString method* | |

This approach does not lend itself to easy code maintenance since adding further subclasses of BankAccounts to our example requires additional modifications to the toString method of AccountsLedger. For each type of account, we would need to extend the toString method to check for the account type and build the necessary String representation of the class. What if we could write a single line of code that calls the toString method for each object in the AccountsLedger so that the system automatically calls the appropriate class' method at run-time? The toString method could then look like the following:

|  |  |
| --- | --- |
| 1:  2:  3:  4:  5:  6:  7:  8:  9: 10: 11: 12: 13: 14: 15: 16: 17: 18: 19: 20: | /\*\*  \* Return a string representation of the account ledger.  \*  \* @return - String for each account in the ledger.  \*/  public String toString() {  String accountString = "";  /\*\*  \* For each account in the vector, append the String representation  \* of it to the String named accountString.  \*/  for( int i=0; i < accounts.size(); i++ ) {  accountString = accountString + accounts.get(i).toString()  + "\n";  }  return accountString;  } |
| **Listing 3** *AccountsLedger toString method* | |

Polymorphism allows us to do just this. Polymorphism allows us to call the toString method for each account in the accounts Vector. The Java interpreter determines what type the current element in the Vector named accounts is at run-time and automatically calls the toString method for that element's type. The CheckingAccount and SavingsAccount classes can each implement their own toString method, computing and returning class-specific information. We can do this because the toString method is defined for all classes — all classes in Java extend Object, which defines toString.

The toString method for the CheckingAccount class is below:

|  |  |
| --- | --- |
| 1:  2:  3:  4:  5:  6:  7:  8:  9: 10: 11: 12: 13: 14: | /\*\*  \* Return a string representation of this account.  \*  \* @return String - string of all account info  \*/  public String toString() {  /\*\*  \* Call the getName accessor method of the BankAccount  \* class to return the name of the account. The balance  \* is protected, so we can access it directly.  \*/  return new String( "Checking account: " + getName()  + " balance: $ " + balance );  } |
| **Listing 4** *CheckingAccount toString method* | |

The toString method for the SavingsAccount class is below:

|  |  |
| --- | --- |
| 1:  2:  3:  4:  5:  6:  7:  8:  9: 10: 11: 12: 13: 14: 15: | /\*\*  \* Return a string representation of this account.  \*  \* @return - a string containing information about this account.  \*/  public String toString() {  /\*\*  \* Call the getName accessor method of the BankAccount  \* class to return the name of the account. The balance  \* is protected, so we can access it directly.  \*/  return new String( "Savings account : " + getName()  + " balance: $ " + balance + " earning "  + interestRate + " % interest");  } |
| **Listing 5** *SavingsAccount toString method* | |

Using a feature of Java known as overriding methods, which we will soon discuss, we can then implement the toString method of the AccountsLedger class as we have proposed above.

You may have noticed an unusual character sequence in the AccountsLedger's toString method. The "\n" character sequence represents the newline character. The newline character causes all text after it to be printed on a new line. For example, consider the following string of text:

"This is printed on line one.\nAnd this is printed on line two."

If Java were to output this string of text, it would look like:

This is printed on line one.

And this is printed on line two.

The applet below simulates adding three different accounts to the AccountsLedger class and calling the toString method. One account is owned by Alice. One account is owned by Bob. The final account is owned by Chuck.

**Downcasting**

Calling a method polymorphically does not work if that method is not defined for all types that the reference can take. Let us examine a scenario where we wanted to change the interest rate for each account in a ledger of SavingsAccount objects to 3%. We could not call the setInterestRate method for each account in ledger as above. Recall that Vector objects store instances of the Object class. When the get method of a Vector is called, it returns a reference to an Object. Since the setInterestRate method is not defined for an Object instance, we cannot call it as follows:

|  |  |
| --- | --- |
| 1: 2: 3: | for(int i = 0; i < accounts.size(); i++) {  accounts.get(i).setInterestRate(3.00);  } |
| **Listing 6** *Setting the interest rate of all accounts* | |

We must cast the return value of the get method to a reference of type SavingsAccount. In particular, must *downcast* the reference. Downcasting a reference casts it from a more general type to a more specific type. In this case, the reference is cast to a SavingsAccount, which is a more specific form of an Object. Once the reference is cast, we can then call the setInterestRate method because the reference is now of type SavingsAccount, for which the setInterestRate method is defined. This is illustrated by the following implementation.

|  |  |
| --- | --- |
| 1:  2:  3:  4:  5:  6:  7:  8:  9: 10: 11: 12: 13: | for(int i = 0; i < accounts.size(); i++) {  /\*\*  \* Downcast the return value of get to a SavingsAccount.  \*/  SavingsAccount savings = (SavingsAccount) accounts.get(i);  /\*\*  \* Set the interest rate. This will  \* immediately change the version stored in the vector.  \*/  savings.setInterestRate(3.00);  } |
| **Listing 7** *Setting the interest rate of all accounts* | |

You should note that the above code will work only if the Vector contains objects of the same type. If the Vector contained objects of different types, a run-time error would occur.

**Overriding Methods**

In [3.1 Introduction to Inheritance](javascript:ContentByName('pg-introtoinheritance');), you saw that a subclass acquires all methods of a parent class via inheritance. A subclass may re-implement the methods of a parent's class. A method is *overridden* when the parent class' method signature is maintained but the implementation changes. A method's *signature* includes its name and its parameter list. For example, the signature of the withdraw method of BankAccount is:

withdraw(double amount)

Overriding methods allows polymorphism to work. Recall the example toString method for the AccountsLedger class. The version of the toString method that used polymorphism worked because the toString method was overridden for the SavingsAccount and CheckingAccount classes.

Polymorphism is not the only reason to override methods. It may make sense for a subclass to share the signature of a method with its parent class given the semantics of the application, even though the subclass may implement the method differently. Consider a checking account with an upper limit on the number of checks that may be written. If the number of checks written exceeds a set threshold (ten for example) then a one-dollar fee is charged for each withdrawal. This may be implemented by overriding the withdraw method in the CheckingAccount class. Consider the overridden form of withdraw below.

|  |  |
| --- | --- |
| 1:  2:  3:  4:  5:  6:  7:  8:  9: 10: 11: 12: 13: 14: 15: 16: 17: | /\*\*  \* Overridden withdraw method.  \*  \* @param amount - amount of money to withdraw  \*/  public void withdraw(double amount) {  balance = balance - amount;  numChecksWritten = numChecksWritten + 1;  /\*\*  \* If more than ten checks have been written against this  \* account, charge a one dollar penalty.  \*/  if ( numChecksWritten >10 ) {  balance = balance - 1.00;  }  } |
| **Listing 8** *CheckingAccount withdraw method* | |

You should not change the intended meaning of an overridden method. If you intend to override a method in a sub class, it should still have the same general effect as the method of the super class. For example, consider the getName method of BankAccount. This method is intended to return a String representing the name of a bank account. This method, if overridden in subclasses, should perform the same operation.

**The super Keyword**

Overriding allows you to change the way a subclass implements a method. When overriding a method, you may also find yourself extending its functionality instead of completely re-implementing it. Thus, you are copying the same code in the superclass' implementation of a method. The super keyword allows us to invoke a parent class' version of a method, including a constructor, which saves you from having to copy a parent class' code!

Recall that the name instance variable of BankAccount is private. Thus, SavingsAccount and CheckingAccount cannot set its value using their constructors. However, the constructors can set the balance as it is protected. It would be convenient for programmers if the SavingsAccount and CheckingAccount classes allowed their constructors to set the same information as the BankAccount constructor.

You may be asking yourself "How do I set the name in the subclass constructor when the instance variable is private?" One potential way to set the value is to write a method in the BankAccount class that sets the name. This solution does not generalize well for classes with large numbers of private instance variables that are initialized by the constructor, since a method must be written that sets each instance variable.

Java allows a superclass' method to be called during overriding by using the super keyword. Consider an implementation of the SavingsAccount class's constructor where, in addition to specifying an account name and balance, we would also like to specify a special interest rate to use instead of the standard bank interest rate, which is set at 2% for in the SavingsAccount class. This may be implemented using the super keyword. We can use the super keyword to call the parent class's constructor. The super keyword may be used to set the name and balance parameters of the BankAccount class from within the SavingsAccount constructor. The code that implements this constructor follows:

|  |  |
| --- | --- |
| 1:  2:  3:  4:  5:  6:  7:  8:  9: 10: 11: 12: 13: | /\*\*  Constructor calls the BankAccount class constructor  in addition to setting the new interest rate.  \*  @param initialName - name of the account  @param initialBalance - balance of the account  @param initialInterestRate - interest rate for the account.  \*/  public SavingsAccount(String initialName, double initialBalance,  double initialInterestRate ) {  super(initialName, initialBalance);  interestRate = initialInterestRate;  } |
| **Listing 9** *SavingsAccount constructor* | |

Note that if you are using the super keyword in a constructor, Java convention states that you should use super before you call any other statements.

The super keyword may be used to access methods other than constructors. Recall the overridden withdraw method of the CheckingAccount class above. Instead of deducting the input amount in the CheckingAccount's method, we could have used the super keyword to call BankAccount's withdraw method. Therefore, the withdraw method could look like the following:

|  |  |
| --- | --- |
| 1:  2:  3:  4:  5:  6:  7:  8:  9: 10: 11: 12: 13: | public void withdraw(double amount) {  numChecksWritten = numChecksWritten + 1;  /\*\*  \* If the number of checks written is greater than ten,  \* charge a $1.00 fee.  \*/  if ( numChecksWritten >10 ) {  amount = amount + 1.00;  }  super.withdraw(amount);  } |
| **Listing 10** *Setting the interest rate of all accounts* | |

**The Complete Source Code**

The complete source code for the BankAccount, CheckingAccount, SavingsAccount, and AccountsLedger classes are as follows:

The BankAccount class:

|  |  |
| --- | --- |
| 1:  2:  3:  4:  5:  6:  7:  8:  9: 10: 11: 12: 13: 14: 15: 16: 17: 18: 19: 20: 21: 22: 23: 24: 25: 26: 27: 28: 29: 30: 31: 32: 33: 34: 35: 36: 37: 38: 39: 40: 41: 42: 43: 44: 45: 46: 47: 48: 49: 50: 51: 52: 53: 54: 55: 56: 57: 58: 59: 60: 61: 62: 63: 64: 65: | /\*\*  \* Class modeling a simple BankAccount.  \*  \* @author iCarnegie vp  \*/  public class BankAccount {  /\*\*  \* BankAccount's balance  \*/  protected double balance = 0.0;  /\*\*  \* BankAccount's name  \*/  private String name;  /\*\*  \* BankAccount's constructor takes initialName and initialBalance  \* as input.  \*  \* @param initialName - name of the account  \* @param initialBalance - balance of the account  \*/  public BankAccount(String initialName, double initialBalance) {  name = initialName;  balance = initialBalance;  }  /\*\*  \* Deposit money into the account.  \*  \* @param amount - amount of money to deposit.  \*/  public void deposit(int amount) {  balance = balance + amount;  }  /\*\*  \* Withdraw money from an account.  \*  \* @param amount - amount of money to withdraw.  \*/  public void withdraw(int amount) {  balance = balance - amount;  }  /\*\*  \* Return the name of the account.  \*  \* @return - a string representing the account name.  \*/  public String getName() {  return name;  }  /\*\*  \* Return the balance of the account.  \*  \* @return - a double representing the account balance.  \*/  public double getBalance() {  return balance;  }  } |
| **Listing 11** *[The BankAccount class](http://www.icarnegie.com/content/SSD/SSD1/2.1/normal/pg-inheritance/pg-usinginheritance/BankAccount.java" \t "external)* | |

The CheckingAccount class:

|  |  |
| --- | --- |
| 1:  2:  3:  4:  5:  6:  7:  8:  9: 10: 11: 12: 13: 14: 15: 16: 17: 18: 19: 20: 21: 22: 23: 24: 25: 26: 27: 28: 29: 30: 31: 32: 33: 34: 35: 36: 37: 38: 39: 40: 41: 42: 43: 44: 45: 46: 47: 48: 49: 50: 51: 52: 53: 54: 55: 56: | /\*\*  \* A class representing a checking account.  \*  \* @author iCarnegie vp  \*/  public class CheckingAccount extends BankAccount {  /\*\*  \* The number of checks written against this account.  \*/  private int numChecksWritten = 0;  /\*\*  \* Constructor calls the BankAccount constructor via super.  \*  \* @param initialName - name of the account  \* @param initialBalance - balance of the account  \*/  public CheckingAccount(String initialName, double initialBalance) {  super(initialName, initialBalance);  }  /\*\*  \* Overridden withdraw method.  \*  \* @param amount - amount of money to withdraw  \*/  public void withdraw(double amount) {  balance = balance - amount;  numChecksWritten = numChecksWritten + 1;  /\*\*  \* If more than ten checks have been written against this  \* account, charge a one dollar penalty.  \*/  if ( numChecksWritten >10 ) {  balance = balance - 1.00;  }  }  /\*\*  \* Return a string representation of this account.  \*  \* @return String - string of all account info  \*/  public String toString() {  /\*\*  \* Call the getName accessor method of the BankAccount  \* class to return the name of the account. The balance  \* is protected, so we can access it directly.  \*/  return new String( "Checking account: " + getName()  + " balance: $ " + balance );  }  } |
| **Listing 12** *[The CheckingAccount class](http://www.icarnegie.com/content/SSD/SSD1/2.1/normal/pg-inheritance/pg-usinginheritance/CheckingAccount.java" \t "external)* | |

The SavingsAccount class:

|  |  |
| --- | --- |
| 1:  2:  3:  4:  5:  6:  7:  8:  9: 10: 11: 12: 13: 14: 15: 16: 17: 18: 19: 20: 21: 22: 23: 24: 25: 26: 27: 28: 29: 30: 31: 32: 33: 34: 35: 36: 37: 38: 39: 40: 41: 42: 43: 44: 45: 46: 47: 48: 49: 50: 51: 52: 53: 54: 55: 56: 57: 58: 59: 60: 61: | /\*\*  \* Class representing a SavingsAccount.  \*  \* @author iCarnegie vp  \*/  public class SavingsAccount extends BankAccount {  /\*\*  \* Each savings account has an interest which defaults to 2%.  \*/  private double interestRate = 2.0;  /\*\*  \* Constructor calls the BankAccount class constructor  \* in addition to setting the new interest rate.  \*  \* @param initialName - name of the account  \* @param initialBalance - balance of the account  \* @param initialInterestRate - interest rate for the account.  \*/  public SavingsAccount(String initialName, double initialBalance,  double initialInterestRate ) {  super(initialName, initialBalance);  interestRate = initialInterestRate;  }  /\*\*  \* Return this account's interest rate.  \*  \* @return - double representing the interest rate.  \*/  public double getInterestRate() {  return interestRate;  }  /\*\*  \* Set this account's interest rate.  \*  \* @param - double representing the new interest rate.  \*/  public void setInterestRate(double newRate) {  interestRate = newRate;  }  /\*\*  \* Return a string representation of this account.  \*  \* @return - a string containing information about this account.  \*/  public String toString() {  /\*\*  \* Call the getName accessor method of the BankAccount  \* class to return the name of the account. The balance  \* is protected, so we can access it directly.  \*/  return new String( "Savings account : " + getName()  + " balance: $ " + balance + " earning "  + interestRate + " % interest");  }  } |
| **Listing 13** *[The SavingsAccount class](http://www.icarnegie.com/content/SSD/SSD1/2.1/normal/pg-inheritance/pg-usinginheritance/SavingsAccount.java" \t "external)* | |

The AccountsLedger class:

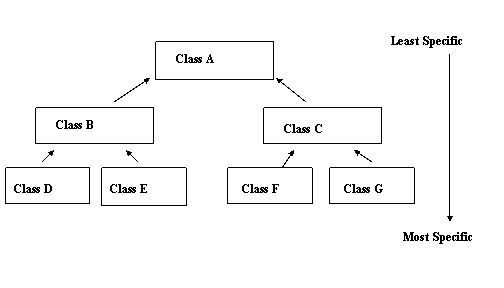
|  |  |
| --- | --- |
| 1:  2:  3:  4:  5:  6:  7:  8:  9: 10: 11: 12: 13: 14: 15: 16: 17: 18: 19: 20: 21: 22: 23: 24: 25: 26: 27: 28: 29: 30: 31: 32: 33: 34: 35: 36: 37: 38: 39: 40: 41: 42: 43: 44: 45: 46: 47: 48: 49: 50: 51: 52: 53: | import java.util.\*;  /\*\*  \* Class representing a collection, or ledger, of accounts.  \*  \* @author iCarnegie vp  \*/  public class AccountsLedger {  /\*\*  \* Bank accounts are stored in a vector.  \*/  private Vector accounts;  /\*\*  \* Constructor creates a new vector.  \*/  public AccountsLedger() {  accounts = new Vector();  }  /\*\*  \* Return a string representation of the account ledger.  \*  \* @return - String for each account in the ledger.  \*/  public String toString() {  String accountString = "";  /\*\*  \* For each account in the vector, append the String representation  \* of it to the String named accountString.  \*/  for( int i=0; i < accounts.size(); i++ ) {  accountString = accountString + accounts.get(i).toString()  + "\n";  }  return accountString;  }  /\*\*  \* Adds a new account to the ledger. This form can take  \* any BankAccount, or sub class of BankAccount, as input.  \*  \* @param account - account to add to the ledger.  \*/  public void addAccount(BankAccount account) {  accounts.add(account);  }  } |
| **Listing 14** *[The AccountsLedger class](http://www.icarnegie.com/content/SSD/SSD1/2.1/normal/pg-inheritance/pg-usinginheritance/AccountsLedger.java" \t "external)* | |

**3.3 Designing a Class Hierarchy**

**What Is Designing a Class Hierarchy?**

The process of designing a class hierarchy allows a programmer to transform a set of related classes into a hierarchical categorization of classes. You can think of a class hierarchy as a tree. Classes at the root (top) of the tree are the most general. Classes that are leaves, that is, at the bottom of the tree, are more specific. Classes at lower levels of the tree are more specialized forms of the classes at the higher levels, with more customized behaviors and data.

Consider the three-level class hierarchy tree below. Seven classes are defined in this hierarchy, Classes A, B, C, D, E, F, and G. Class A is the root and is the most general of all the classes; it is the superclass for the entire hierarchy. Classes B and C are in the middle of the hierarchy and are more specific forms of Class A. Classes D, E, F, and G are leaf nodes, and are therefore the most specific classes in the hierarchy. Classes D and E are the most specific forms of the superclass, Class B. Classes F and G are the most specific forms of their superclass, Class C.



**The Design Process**

The process of designing a class hierarchy has a number of steps. First, you must examine all of the classes you intend to categorize for common attributes and methods. Second, you should identify an appropriate superclass structure using the information you have gathered in the first step. Third, now that your hierarchy is in place, you should move the common characteristics from the subclass to the superclass. Part of this process is setting the appropriate access modifiers for attributes and methods. Finally, you should look at the overall hierarchy and see if there are any characteristics that can be moved across the different classes in the hierarchy.

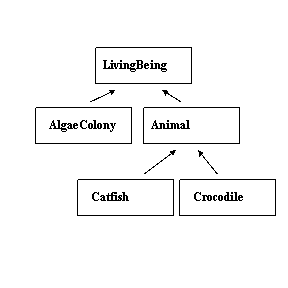
The process of taking an existing group of classes and molding them into a hierarchy using inheritance is sometimes referred to as *refactoring*. The process of refactoring classes involves reviewing the classes and determining how they relate to one another. In particular, the process searches for unexploited uses of the "is-a" relationship.

We should note that software designers often do not design complete classes, including all of the details for each class, first, and then assign them to a hierarchy. The design process typically goes in the other direction; a general hierarchy is established first, and then the details are established and factored into each class. This runs counter to the method we have just followed in designing the AlgaeColony, Catfish, and Crocodile classes. We started with the details of each class and then organized them into a hierarchy. We have done this in SSD1 for pedagogical reasons.

**The Simulation Class Hierarchy**

Let us now examine the process of designing a class hierarchy in the context of the simulation. By now, you have already coded the AlgaeColony, Catfish, and Crocodile classes. The classes you have coded already extend a class named LivingBeing, which was left empty. When coding these classes, you added much of the same code in all three classes. We have pointed out that duplicating this code is wasteful. We will now examine the transformation of these classes into a hierarchical structure in greater detail using the steps outlined above.

First, let us examine the three classes. As we mentioned previously, algae colonies, catfish, and crocodiles are all living beings. Catfish and crocodiles are animals and share characteristics that algae colonies do not. It makes sense, therefore, to move the attributes and behavior shared by all living beings into the LivingBeing class. We will also create a class named Animal that encapsulates the shared methods and attributes of the Catfish and Crocodile classes. Animal is a subclass of LivingBeing. The figure below is a pictorial representation of this hierarchy.



We must identify the common attributes and behaviors that must be subsumed into the LivingBeing class. The following instance variables are common to the AlgaeColony, Catfish, and Crocodile classes.

* *row*. Row in which the living being is located
* *column*. Column in which the living being is located
* *deadOrAlive*. Is the living being dead or alive
* *energy*. Amount of energy the living being has
* *age*. Age expressed as blocks of time lived
* *name*. Name of this living being
* *simulation*. The simulation to which this living being belongs
* *minEnergy*. Minimum energy level needed to survive
* *maxEnergy*. Maximum energy level that the living being could carry

The following class variables are common to the AlgaeColony, Catfish, and Crocodile classes

* *ALIVE*. The value "alive" indicating the being is still alive
* *DEAD*. The value "dead" indicating the being is dead

The following methods are common to the AlgaeColony, Catfish, and Crocodile classes.

* *getRow*. Get the row at which the living being is located
* *getColumn*. Get the column at which the living being is located
* *getAge*. Get the living being's age
* *getColor*. Get the color of the living being expressed in hex notation
* *getName*. Get the name of this living being
* *getImage*. Get the filename that contains the image of the living being
* *getDisplayMechanism*. Get the preferred display mechanism for living being
* *getMinEnergy*. Get the minimum energy needed to live
* *getMaxEnergy*. Get the maximum energy that the living being can carry
* *getSpecies*. Returns the species
* *getEnergy*. Get the energy currently carried by the living being
* *setEnergy*. Sets energy level
* *die*. Change the deadOrAlive to DEAD
* *isDead*. Is the living being dead
* *liveALittle*. The living being's age increases by one time block

Now that we have identified common attributes and behaviors of the AlgaeColony, Catfish, and Crocodile classes, we must identify the common attributes and behaviors of the Catfish and Crocodile classes. The following class variables are common to the Catfish and Crocodile classes.

* *RIGHT*. Represents moving right
* *LEFT*. Represents moving left
* *UP*. Represents moving up
* *DOWN*. Represents moving down

The following methods are common to the Catfish, and Crocodile classes.

* *isHungry*. Is the animal hungry
* *getDirection*. Get the direction faced by the animal
* *moveToRow*. Move the animal to a new row, if new row is within lake bounds
* *moveToColumn*. Move the animal to a new column, if new column is within lake bounds

Next, we must move the shared attributes and behaviors out of the subclasses and into the superclasses. Part of this process is setting the appropriate access modifiers for the methods and variables. Let us assign the access to one method and one instance variable.

The liveALittle method should be set to protected access in the LivingBeing class because subclasses must be able to access it. Recall that the liveALittle method contains one line of code that is shared by all classes. All classes must increment their age. Note from the implementation of the LivingBeing class below that the liveALittle method contains only the line that increments that age. This behavior must be accessible to all subclassses that intend to override the liveALittle method.

The instance variable named direction of the Animal class is changed only by the moveToRow and moveToColumn methods of the Animal class. No other classes can change this variable, nor should they be able to. For this reason, direction should be set to private access.

**The LivingBeing Class**

The complete source code of the LivingBeing class is below.

|  |  |
| --- | --- |
| 1:   2:   3:   4:   5:   6:   7:   8:   9:  10:  11:  12:  13:  14:  15:  16:  17:  18:  19:  20:  21:  22:  23:  24:  25:  26:  27:  28:  29:  30:  31:  32:  33:  34:  35:  36:  37:  38:  39:  40:  41:  42:  43:  44:  45:  46:  47:  48:  49:  50:  51:  52:  53:  54:  55:  56:  57:  58:  59:  60:  61:  62:  63:  64:  65:  66:  67:  68:  69:  70:  71:  72:  73:  74:  75:  76:  77:  78:  79:  80:  81:  82:  83:  84:  85:  86:  87:  88:  89:  90:  91:  92:  93:  94:  95:  96:  97:  98:  99: 100: 101: 102: 103: 104: 105: 106: 107: 108: 109: 110: 111: 112: 113: 114: 115: 116: 117: 118: 119: 120: 121: 122: 123: 124: 125: 126: 127: 128: 129: 130: 131: 132: 133: 134: 135: 136: 137: 138: 139: 140: 141: 142: 143: 144: 145: 146: 147: 148: 149: 150: 151: 152: 153: 154: 155: 156: 157: 158: 159: 160: 161: 162: 163: 164: 165: 166: 167: 168: 169: 170: 171: 172: 173: 174: 175: 176: 177: 178: 179: 180: 181: 182: 183: 184: 185: 186: 187: 188: 189: 190: 191: 192: 193: 194: 195: 196: 197: 198: 199: 200: 201: 202: 203: 204: 205: 206: 207: 208: 209: 210: 211: 212: 213: 214: 215: 216: 217: 218: 219: 220: 221: 222: 223: 224: 225: 226: 227: 228: 229: 230: 231: 232: 233: 234: 235: 236: 237: 238: 239: 240: 241: 242: 243: 244: 245: 246: 247: 248: 249: 250: 251: 252: 253: 254: 255: 256: 257: 258: 259: 260: 261: 262: 263: 264: 265: 266: 267: 268: 269: 270: 271: 272: 273: 274: 275: 276: 277: 278: 279: 280: 281: 282: 283: 284: 285: 286: 287: 288: 289: 290: 291: 292: 293: 294: 295: 296: 297: 298: 299: 300: 301: 302: 303: 304: 305: 306: 307: 308: 309: 310: 311: | /\*  \* Created on Jul 4, 2003  \*  \*/  /\*\*  \* Each living being lives in a location (row, column) at any given time.  \* More than one living being can be in a location. E.g.: algae and fish.  \* Depending on the species, living beings could move.  \* <p>  \* Each living being has some energy level. If the energy level falls  \* below the minimum needed to live, it dies. When a living being  \* dies, it is deleted from the simulation.  \* </p>  \*  \* @author iCarnegie av  \*/  public class LivingBeing {  /\*\*  \* The living being is born "alive".  \* Then it dies, becoming a corpse.  \*/  protected static final String ALIVE = "alive";  /\*\*  \* The living being is born "alive".  \* Then it dies, becoming a "dead" corpse.  \*/  protected static final String DEAD = "dead";  /\*\*  \* Row-wise location of the living being  \*/  protected int row;  /\*\*  \* Column-wise location of the living being  \*/  protected int column;  /\*\*  \* Is the living being dead or alive?  \*/  private String deadOrAlive;  /\*\*  \* Amount of energy the living being has.  \*/  private int energy;  /\*\*  \* Age expressed as blocks of time lived  \*/  private int age = 0;  /\*\*  \* Name of this living being.  \*/  private final String name;  /\*\*  \* The simulation to which this living being belongs.  \* This is needed so the living being can send a message  \* to simulation and ask  \* for prey (or predator) in the neighboring locations.  \* Prey is food. Food is good!  \*/  protected Simulation simulation;  /\*\*  \* Minimum energy level needed to survive.  \* The minimum could increase as the individual grows.  \*/  protected int minEnergy;  /\*\*  \* Maximum energy level that the living being could carry.  \* The maximum could change as the individual grows.  \*/  protected int maxEnergy;  /\*\*  \* Create a living being at a given location with a  \* given energy and store the simulation to which the living being  \* belongs. Cap row and column within lake boundary.  \*  \* @param initialRow - the row location of living being  \* @param initialColumn - the column location of living being  \* @param initialSimulation - the simulation to which being belongs  \* @param initialName - name of the living being  \* @param initialMinEnergy - minimum energy to survive  \* @param initialMaxEnergy - max energy the living being can carry  \*/  protected LivingBeing(  int initialRow,  int initialColumn,  Simulation initialSimulation,  String initialName,  int initialMinEnergy,  int initialMaxEnergy) {  simulation = initialSimulation;  deadOrAlive = ALIVE;  // Set the Row within bounds  if (initialRow > simulation.getLastRow()) {  row = simulation.getLastRow();  } else if (initialRow < simulation.getFirstRow()) {  row = simulation.getFirstRow();  } else {  row = initialRow;  }  // Set the Column within bounds  if (initialColumn > simulation.getLastColumn()) {  column = simulation.getLastColumn();  } else if (initialColumn < simulation.getFirstColumn()) {  column = simulation.getFirstColumn();  } else {  column = initialColumn;  }  // Set the minEnergy and maxEnergy  minEnergy = initialMinEnergy;  maxEnergy = initialMaxEnergy;  energy = simulation.getRand().nextInt(maxEnergy - minEnergy)  + minEnergy;  age = 0;  name = initialName;  }  /\*\*  \* Create an organism that belongs to a specified species at  \* a specified location and add it to the specified simulation.  \*  \* @param simulation - the simulation this organism belongs to  \* @param species - the organism's species  \* @param value - row and column values from HTML Form.  \* row value is the last two digits,  \* column value is given in the remaining digits.  \*/  public static void createLivingBeing(  Simulation sim,  String species,  String value) {  int rowAndCol = Integer.parseInt(value);  int row = rowAndCol / 100;  int column = rowAndCol - (100 \* row);  if (species.equals("algae")) {  sim.addLivingBeing(new AlgaeColony(row, column, sim));  }  if (species.equals("catfish")) {  sim.addLivingBeing(new Catfish(row, column, sim));  }  if (species.equals("crocodile")) {  sim.addLivingBeing(new Crocodile(row, column, sim));  }  }  /\*\*  \* Get the row at which the living being is located  \*  \* @return - the row of the living being's location.  \*/  public int getRow() {  return row;  }  /\*\*  \* Get the column at which the living being is located  \*  \* @return - the column of the living being's location.  \*/  public int getColumn() {  return column;  }  /\*\*  \* Get the living being's age  \*  \* @return the age of the living being expressed in blocks of time  \*/  public int getAge() {  return age;  }  /\*\*  \* Color of the living being expressed in hex notation.  \* For example, the "green-est" color is "#00FF00",  \* "blue-est" is "#0000FF", the "red-est" is "#FF0000".  \*  \* @return the rgb color in hex notation. preceded by a '#'  \*/  public String getColor() {  return "#FFFFFF"; // default is white.  }  /\*\*  \* Get the name of this living being  \*  \* @return the name of the living being.  \*/  public String getName() {  return name;  }  /\*\*  \* Get the filename that contains the image of the living being  \*  \* @return the image of the living being.  \*/  public String getImage() {  return "/blank.gif";  }  /\*\*  \* Get the preferred display mechanism for living being  \*  \* @return one of the constants IMAGE or COLOR,  \* depending on the display mechanism for the living being.  \*/  public String getDisplayMechanism() {  return Simulation.IMAGE;  }  /\*\*  \* Get the minimum energy needed to live.  \*  \* @return the minimum energy needed for the living being to live.  \*/  protected int getMinEnergy() {  return minEnergy;  }  /\*\*  \* get the maximum energy that the living being can carry.  \*  \* @return the maximum energy the living being can carry.  \*/  protected int getMaxEnergy() {  return maxEnergy;  }  /\*\*  \* Returns the species.  \*  \* @return the species  \*/  public String getSpecies() {  return "Unknown";  }  /\*\*  \* Get the energy currently carried by the living being.  \*  \* @return current energy level of the organism  \*/  public int getEnergy() {  return energy;  }  /\*\*  \* Sets energy level. If new energy level is less  \* than minimum energy level, the organism dies.  \* New energy level is capped at maximum energy level.  \*/  protected void setEnergy(int newEnergy) {  if (newEnergy < getMinEnergy()) {  energy = newEnergy;  die();  } else if (newEnergy > getMaxEnergy()) {  energy = getMaxEnergy();  } else {  energy = newEnergy;  }  }  /\*\*  \* Die: Change the deadOrAlive to DEAD.  \*/  protected void die() {  deadOrAlive = DEAD;  }  /\*\*  \* Is the living being dead?  \*  \* @return <code>true</code> if dead. <code>false</code>, otherwise.  \*/  public boolean isDead() {  return (deadOrAlive == DEAD);  }  /\*\*  \* The living being's age increases by one time block.  \*/  public void liveALittle() {  age = age + 1;  }  } |
| **Listing 1** *[The LivingBeing class](http://www.icarnegie.com/content/SSD/SSD1/2.1/normal/pg-inheritance/pg-designingaclasshierarchy/LivingBeing.java" \t "external)* | |

**The Animal Class**

The complete source code of the Animal class follows.

|  |  |
| --- | --- |
| 1:   2:   3:   4:   5:   6:   7:   8:   9:  10:  11:  12:  13:  14:  15:  16:  17:  18:  19:  20:  21:  22:  23:  24:  25:  26:  27:  28:  29:  30:  31:  32:  33:  34:  35:  36:  37:  38:  39:  40:  41:  42:  43:  44:  45:  46:  47:  48:  49:  50:  51:  52:  53:  54:  55:  56:  57:  58:  59:  60:  61:  62:  63:  64:  65:  66:  67:  68:  69:  70:  71:  72:  73:  74:  75:  76:  77:  78:  79:  80:  81:  82:  83:  84:  85:  86:  87:  88:  89:  90:  91:  92:  93:  94:  95:  96:  97:  98:  99: 100: 101: 102: 103: 104: 105: 106: 107: 108: 109: 110: 111: 112: 113: 114: 115: 116: 117: 118: 119: 120: 121: 122: 123: 124: 125: 126: 127: 128: 129: 130: 131: 132: 133: 134: 135: 136: 137: 138: 139: 140: 141: 142: 143: 144: 145: 146: 147: 148: 149: 150: 151: 152: 153: 154: 155: 156: 157: 158: 159: 160: | /\*  \* Created on Jul 6, 2003  \*  \*/  // Concept example: extends.  // Each Animal is a LivingBeing.  // Animal should inherit all attributes and behavior of LivingBeing.  /\*\*  \* Animals can move.  \*  \* @author iCarnegie av  \*  \*/  public class Animal extends LivingBeing {  // Concept example: final. since it is a constant  // Concept example: static. since only one value is needed  // irrespective of number of object instances  /\*\*  \* String constant - used to indicate the direction animal is facing.  \*/  protected static final String RIGHT = "right";  /\*\*  \* String constant - used to indicate the direction animal is facing.  \*/  protected static final String LEFT = "left";  /\*\*  \* String constant - used to indicate the direction animal is facing.  \*/  protected static final String UP = "up";  /\*\*  \* String constant - used to indicate the direction animal is facing.  \*/  protected static final String DOWN = "down";  /\*\*  \* Which direction am I facing.  \*/  private String direction;  /\*\*  \* Create an animal at a given location with a  \* given energy and store the simulation to which the animal belongs.  \* Cap row and column within lake boundary.  \*  \* @param initialRow - the row location of animal  \* @param initialColumn - the column locaiton of animal  \* @param initialSimulation - the simulation to which animal belongs  \* @param initialName - name of the animal  \* @param initialMinEnergy - minimum energy to survive  \* @param initialMaxEnergy - max energy the animal can carry  \*/  public Animal(  int initialRow,  int initialColumn,  Simulation initialSimulation,  String initialName,  int minEnergy,  int maxEnergy) {  super(  initialRow,  initialColumn,  initialSimulation,  initialName,  minEnergy,  maxEnergy);  direction = RIGHT; // Start by facing east.  }  /\*\*  \* Get the direction faced by the animal.  \*  \* @return the facing direction.  \*/  protected String getDirection() {  return direction;  }  /\*\*  \* Is the animal hungry?  \*  \* @return True, if hungry. False, otherwise.  \*/  protected boolean isHungry() {  // Hungry, if current energy level is less than twice the  // amount needed for survival.  return (getEnergy() < (2 \* getMinEnergy()));  }  /\*\*  \* Move the animal to a new row, if new row is within lake bounds.  \*  \* @param newRow - the row to move to.  \* @return the row moved to. -1, if dead  \*/  protected int moveToRow(int newRow) {  if (isDead()) {  return -1;  }  // Keep the new value within lake boundary.  if (newRow > simulation.getLastRow()) {  newRow = simulation.getLastRow();  } else if (newRow < simulation.getFirstRow()) {  newRow = simulation.getFirstRow();  }  // I might face a new direction.  if (newRow < row) {  direction = UP;  } else if (newRow > row) {  direction = DOWN;  }  row = newRow;  return row;  }  /\*\*  \* Move the animal to a new column, if column is within lake bounds.  \*  \* @param newColumn - the column to move to.  \* @return the column moved to. Lake boundary limits movement.  \*/  protected int moveToColumn(int newColumn) {  if (isDead()) {  return -1;  }  // System.out.println("newColumn = " + newColumn);  // System.out.flush();  // Keep the new value within lake boundary.  if (newColumn > simulation.getLastColumn()) {  newColumn = simulation.getLastColumn();  } else if (newColumn < simulation.getFirstColumn()) {  newColumn = simulation.getFirstColumn();  }  // I might face a new direction.  if (newColumn < column) {  direction = LEFT;  } else if (newColumn > column) {  direction = RIGHT;  }  column = newColumn;  return column;  }  } |
| **Listing 2** *[The Animal class](http://www.icarnegie.com/content/SSD/SSD1/2.1/normal/pg-inheritance/pg-designingaclasshierarchy/Animal.java" \t "external)* | |

**The AlgaeColony Class**

Now that we have created our two superclasses, let us examine how two of our classes have changed. Consider what we have removed from the AlgaeColony class. We have removed all of the attributes and behaviors that are shared among the AlgaeColony, Catfish, and Crocodile classes. There are still attributes and behaviors that are specific to the AlgaeColony class. The following class variables are specific to the AlgaeColony class:

* *BABY\_MIN\_ENERGY*. Lowest possible energy needed for a baby to survive
* *BABY\_MAX\_ENERGY*. Maximum energy that a baby can store
* *ENERGY\_TO\_LIVE*. Amount of energy needed to live for a block of time
* *SPECIES*. Name of species
* *nAlgaeCreated*. Number of Algae objects created so far

The AlgaeColony class uses one unique method, giveUpEnergy, which makes an AlgaeColony loose energy when it is being eaten. In addition to this method, AlgaeColony must override a number of methods from the LivingBeing class. These methods are as follows:

* getDisplayMechanism
* getSpecies
* getColor
* liveALittle

The complete code of the AlgaeColony class is as follows:

|  |  |
| --- | --- |
| 1:   2:   3:   4:   5:   6:   7:   8:   9:  10:  11:  12:  13:  14:  15:  16:  17:  18:  19:  20:  21:  22:  23:  24:  25:  26:  27:  28:  29:  30:  31:  32:  33:  34:  35:  36:  37:  38:  39:  40:  41:  42:  43:  44:  45:  46:  47:  48:  49:  50:  51:  52:  53:  54:  55:  56:  57:  58:  59:  60:  61:  62:  63:  64:  65:  66:  67:  68:  69:  70:  71:  72:  73:  74:  75:  76:  77:  78:  79:  80:  81:  82:  83:  84:  85:  86:  87:  88:  89:  90:  91:  92:  93:  94:  95:  96:  97:  98:  99: 100: 101: 102: 103: 104: 105: 106: 107: 108: 109: 110: 111: 112: 113: 114: 115: 116: 117: 118: 119: 120: 121: 122: 123: 124: 125: 126: 127: 128: 129: 130: 131: 132: 133: 134: 135: 136: 137: 138: 139: 140: 141: 142: 143: 144: 145: 146: 147: 148: 149: 150: 151: 152: 153: 154: 155: 156: 157: 158: 159: 160: 161: 162: 163: 164: 165: 166: 167: 168: 169: 170: 171: 172: 173: | /\*  \* Created on Jul 4, 2003  \*  \*/  /\*\*  \* AlgaeColony does not move. If there is sunlight, a portion of the  \* solar energy will be converted into life-energy.  \*  \* @author iCarnegie av  \*/  public class AlgaeColony extends LivingBeing {  /\*\*  \* Lowest possible energy needed for a baby to survive.  \*/  private static final int BABY\_MIN\_ENERGY = 5;  /\*\*  \* Maximum energy that a baby can store.  \*/  private static final int BABY\_MAX\_ENERGY = 255;  /\*\*  \* Amount of energy needed to live for a block of time.  \*/  private static final int ENERGY\_TO\_LIVE = 1;  /\*\*  \* Name of species  \*/  private static final String SPECIES = "Algae";  /\*\*  \* Number of Algae objects created so far.  \*/  private static int nAlgaeCreated = 0;  /\*\*  \* Constructor. Initialize an algae to start life at a specified  \* location with a specified energy. If location is out of bounds,  \* locate the algae at the nearest edge.  \*  \* @param initialRow - the row at which the algae is located  \* @param initialColumn - the column at which the algae is located  \* @param initialSimulation - the simulation that algae belongs to  \*/  public AlgaeColony(  int initialRow,  int initialColumn,  Simulation initialSimulation) {  super(  initialRow,  initialColumn,  initialSimulation,  SPECIES + nAlgaeCreated,  BABY\_MIN\_ENERGY,  BABY\_MAX\_ENERGY);  ++nAlgaeCreated;  }  /\*\*  \* The mechanism to display Algae is to use its color.  \*  \* @return a constant from {@link Simulation#COLOR Simulation} class  \*/  public String getDisplayMechanism() {  return Simulation.COLOR;  }  /\*\*  \* Get the species that the algae belongs to  \*  \* @return a string indicating the species.  \*/  public String getSpecies() {  return SPECIES;  }  /\*\*  \* Get the color of Algae  \*  \* @return - the color as a string in hexademinal notation  \*/  public String getColor() {  // Concept example: Distinction between local variable  // and instance variable.  // Note: Since energyLevel is declared as a \*private\* instance  // in LivingBeing it is not visible here.  // We create a local variable  // whose name happens to be the same as the instance variable.  // The local variable is assigned a value that is the  // instance variable's value.  int energy = getEnergy();  // Let us limit the energy to the color range of 0 to 255.  if (energy < 0) {  energy = 0;  }  if (energy > 255) {  energy = 255;  }  // Convert energy scale into green scale expressed in hex form.  String greenLevel = Integer.toHexString(energy);  // If energy is a value less than 16,  // there will be only one character.  // Since we need 2 characters for green, we pad left with a "0".  if (energy < 16) {  greenLevel = "0" + greenLevel;  }  // Now prepend with "00" to indicate there is no red  // and append with "00" to indicate there is no blue.  return "#00" + greenLevel + "00";  }  /\*\*  \* Algae is being eaten up.  \* So, relinquish energy up to the amount requested.  \* If no energy remains, die.  \*  \* @param energyWanted amount of energy requested - expressed as int.  \* @return - the amount of energy that algae can give up.  \* If the requested energy is greater than the available energy,  \* only the available energy will be given up.  \*/  public int giveUpEnergy(int energyWanted) {  int energy = getEnergy();  // We do not know what it means to want negative energy!!!  if (energyWanted < 0) {  return 0;  }  if (energyWanted < energy) {  setEnergy(energy - energyWanted);  } else // caller is asking for more than energy than available.  // Give only what I have.  // I will die since my energy level falls below minimum.  {  energyWanted = energy;  setEnergy(getMinEnergy() - 1);  }  return energyWanted;  }  /\*\*  \* Algae lives its life. May gain or lose energy.  \*/  public void liveALittle() {  if (isDead()) {  return;  }  super.liveALittle();  int row = getRow();  int column = getColumn();  int sun = simulation.getSunlight(row, column);  // Let us assume that algae can convert 50% of  // solar energy into life-energy.  // Algae has to give up some energy to live.  setEnergy((int) (sun \* 0.5) + getEnergy() - ENERGY\_TO\_LIVE);  }  } |
| **Listing 3** *[The AlgaeColony class](http://www.icarnegie.com/content/SSD/SSD1/2.1/normal/pg-inheritance/pg-designingaclasshierarchy/AlgaeColony.java" \t "external)* | |

**The Catfish Class**

At the bottom of our class hierarchy are the two animals themselves, the Catfish and Crocodile classes. As an exercise, you design a version of Catfish that uses inheritance. Let us take a brief look at this version of the Catfish class to get you started. The Catfish class extends Animal. Catfish must exhibit a number of attributes and behaviors in the form of methods and variables. Some of these attributes and behaviors are already implemented in the LivingBeing or Animal classes. Some behaviors are implemented in the LivingBeing or Animal classes but need to be overridden. Some attributes and behaviors must be implemented specially for the Catfish class. A subset of these attributes and behaviors are listed on [2.2.6 The AlgaeColony Class](javascript:ContentByName('pg-algae-class');). It is up to you to decide which attributes and behaviors are missing from this list and therefore you must implement, which attributes and behaviors can be inherited directly, and which behaviors must be overridden.

* **Guidelines for Designing a Class Hierarchy**

1. Subclasses extend the superclass in a class hierarchy by modifying and adding behavior. For that reason, common behavior should be pushed as high as possible in the class hierarchy. For example, not all LivingBeings can move, and not all Animals can swim. We added the ability to swim to the class Catfish, and the ability to move to the Animal class.
2. Subclasses are specialized forms of superclasses. Therefore, each subclass may require specialized forms of methods implemented in the superclass. You can specialize the behavior of subclasses by overriding methods. Recall that the AlgaeColony, Catfish, and Crocodile classes live their lives differently, and, hence, override the liveALittle method.
3. When deciding between public, private, and protected access for a method or attribute, ask yourself what other classes need access to that method or attribute. Allowing the least amount of access prevents methods from being used in unintended ways. Methods and instance variables used only within a class should be set to private. Methods and instance variables used only within a class and by its subclasses should be set to protected. Public instance variables and methods are visible to all classes and should be used only if it is appropriate to allow access from outside of a class.
4. The meaning of a feature must be maintained from a superclass to a subclass. If you intend to override a method in a subclass, it should still have the same general effect as the method of the superclass. For example, consider the getSpecies method of LivingBeing. This method is intended to return a String representing the name of a species. This method, if overridden in subclasses, should perform the same operation.
5. All features (behaviors and attributes) in a superclass must also apply to a subclass. Features of a superclass not applying to a subclass runs counter to the "is-a" relationship. If a feature of a super class does not apply to one of its subclasses, the subclass should not extend the super class. If you find yourself ignoring a parent class' feature, you should reconsider the class' place in the hierarchy. Recall that the AlgaeColony class does not move and is never hungry. If it were a subclass of Animal, it would need to ignore that functionality.