Module 1.1. Structural versus Algorithmic Aspects

Overview

There are two major aspects of a programming language:

(1) the algorithmic or procedural aspect.

(2) the structural or organizational aspect.

You can make an analogy to a cookbook. Each recipe is a sequence of instructions to carry out in order to cook something. This is the algorithmic aspect.

The recipes may be organized in chapters —appetizers, entrees, desserts, breads, etc. Each chapter may be further subdivided into sections. This is the organizational aspect.

In a programming language, the algorithmic aspect has to do essentially with sequences of assignment statements, conditional statements, loops, and procedure/function calls.

When programming languages were first invented —starting in the 1950s with Fortran and moving on to Algol, Lisp, and Cobol about 1960— the major organizational tool was the subroutine (known today as the function, procedure, or method), which corresponds to a recipe in a cookbook. To add more function to a program, one added more subroutines, each of which did its own thing. This rather flat structure did not and does not scale up very well! Software systems consisting of hundreds or even thousands of subroutines were hard to manage and maintain. Yet, the subroutine continued to be the main structuring mechanism in popular programming languages well into the 1980s.

Various other organizational methods were developed. Many of these were organized around modules, which could contain collections of subroutines and other things. But the organizational method that has had the most success is object-orientation, or OO.

The basics of OO

Here's an analogy that will give the basic ideas of OO. A typical office has filing cabinets. Each drawer contains a bunch of manilla folders. The folders in one drawer have exactly same kind of information. They contain data (e.g. a person's name and address, the amount of money they owe). In addition, for OO, they contain instructions (called methods) for carrying out operations on the data —changing the data, checking the data, performing calculations. Because the instructions have been placed in each manilla folder, anyone in the office can do any job.

In OO, each file drawer is called a class, and each manilla folder in the file drawer is called an object of the class.

In addition:

1. One can write a class definition, which gives the format of the manilla folders of the class —what kinds of data and what instructions each manilla folder will contain.

2. The class definition for file drawer (class) SC can say, hey, let my manilla folders contain the same kinds of data and instructions as those defined for class C, in addition to defining more data and instructions. SC inherits everything from C. For example, a manilla folder for class Cornell undergrad will contain all the information and instructions of class Cornell student plus any information and instructions that pertain only to undergrads.

3. There is a simple way to create a new manilla folder (object) to go in a file drawer (class).

Now you know some essential features of OO. This course CS1130 shows you how these features are realized in the programming language Java.

History of OO

OO has its roots in the language Simula 67, which was developed in the late 1960's. But what was attempted at the time was not fully understood, and its ideas did not catch on. This happens often in science and engineering. Here are other examples. Charles Babbage conceived of computers and designed and tried to build some in the mid 1850s. He didn't have electricity! But the first working computers weren't constructed until the 1940s.

SmallTalk-71 was developed to try out "message-passing" ideas that were in Simula 67, and further explorations were done throughout the 1970s, all at the Xerox Palo Alto Research Center (PARC). In the early 1980's, SmallTalk-80 was shown to the world, and interest in OO grew, leading to OO being incorporated into other languages. The language C++, for example, is an OO-version of C.

Today, OO appears in some guise in a number of languages, like Eiffel, C++, and, of course, Java, to name just a few. Don't expect OO to look exactly the same in all OO languages, but the basic ideas are the same.

DrJava

The Java programming environment used in CS 1130 is DrJava, a free Integrated Development Environment (IDE). DrJava was developed by Professor Corky Cartwright at Rice University, who has developed a number of IDE intended for beginning students. Dr Java is installed in the ACCEL lab and on most on computers in CIT labs. However, you will probably want it on your own computer as well.

If you have problems, and if you have a laptop, the best thing to do is to bring it in to someone to look at. Any of the course staff or consultants can help you with this. If you cannot bring in your computer, we have provided you with some instructions below. If you are having difficulty with these instructions, please email TA Charles Herrmann <cih5@cornell.edu>.

Installing DrJava

Table of Contents

- Installing a Virtual Machine
- Downloading DrJava
- Running DrJava
- DrJava Documentation
- Troubleshooting: Finding tools.jar
- Troubleshooting: Wrong Version Error

Installing a Virtual Machine

DrJava is just a tool to make writing Java source code easier; it does not actually provide you with a copy of the Java software development kit (JDK). So before you do anything else, you need to make sure that you have a copy of (the most recent) JDK.

If you have a Macintosh with Mac OS X, Java should already be installed and you do not have to do anything. Furthermore, as long as you have run the latest Software Update, your version of Java should be the correct one (Java 6 Update 34).

On the other hand, if you are using a PC running a version of Windows, chances are that you don't have the JDK yet. Therefore, you will need to download it from the Java website. If you follow the link, you will see that we are using the JDK 6 Update 34. This is **not** the most recent version of Java (version 7 is more recent). However, it is the most recent version that is guaranteed to work with DrJava. If you run any other version, we cannot guarantee that everything will work correctly. You should choose the version that is appropriate for your version of Windows; this means x64 if you are running 64-bit Windows and x86 if you are running 32-bit Windows.

You should read the instructions when installing the Java platform. One of the things that you will notice is the the download from Oracle gives you both the the Java Runtime

Environment (JRE) AND the Java Development Kit (JDK). The Java Runtime Environment allows you to run Java programs that have already been created (by you or someone else). For example, you need the JRE to run any Java programs that are contained in "jar files". The Java Development Kit, on the other hand, allows you to "compile" Java programs — this means that it translate them into a "machine language" — so they can be run by the Java Runtime Environment.

Downloading DrJava

DrJava is continually being updated. For this class, we will use the most stable version (not the untested **beta** version!), which is currently version 20100913-r5387. The easiest thing to do is to download the version that we provide you. This version has been thoroughly tested with Java 6 update 34, and so is the best one to use.

Alternatively, you can download the latest version of DrJava from the official web page. In addition to the Jar file (which we provide), this site also provides native applications for both Windows and Macintosh. It is fine for you to use the native applications instead of the Jar file; however, you must choose the stable version of DrJava and not the Beta version.

Warning: When downloading, you may get a dialog saying something about a "JAR archive security check". If you do, just right-click on the DrJava download link and select "Save Link As" (or the equivalent in your browser).

Running DrJava

There are several ways to run DrJava. From any command line (on Windows, Macintosh, or Linux), you can type the following:

```
java -jar drjava-stable-20120818-r5686.jar
```

We recommend doing this the first time that you launch DrJava, as the command line will display a lot of helpful error messages if anything goes wrong.

Once you are sure that this works, you are free to use a more graphical approach. On most platforms, simply double-clicking the jar file should start DrJava. However, sometimes double-clicking does not work on Windows machines. The problem may be that some other application has "stolen" the right to be executed when the jar file is doubleclicked. If you are experiencing this problem, visit the JarFix site and see whether it can solve the problem.

If this all sounds too complicated, you can just download the native application for Windows or Macintosh and use that instead.

DrJava Documentation

Once you finally have it running, DrJava is pretty easy to use. The demos given in class and in the demo sessions should help a lot, if you take notes.

If you need more information, help with using DrJava is in Appendix I of the course text and in Chapter 19 of the accompanying CD ProgramLive.

If all else fails, there is documention on using DrJava available on the web.

Troubleshooting: Finding tools.jar

On Windows machines, we have sometimes seen an error that the the system cannot find a file called tools.jar. If this happens, you need to select the correct location of that file. It is typically in a folder like it is in some place like

```
C:\Program Files\Java\jdk1.6.0_29\lib\tools.jar
```

However, the exact folder may differ depending on your version of Java. You can also use a desktop tool to search for this file on your computer.

If you did not set the correct path for the file tools.jar at installation time, set it after installation by going through DrJava's Preferences option. To do so, follow these menu items:

Edit / Preferences / Resource Locations / Tools.jar Location

Then type or browse to the correct location.

Troubleshooting: Wrong Version Error

Do you get a message that looks like this?

```
Error: cannot access javax.swing.JFrame
bad class file: C:\Program
Files\Java\jdk1.6.0_29\lib\rt.jar(javax/swing/JFrame.class)
class file has wrong version 50.0, should be 48.0
Please remove or make sure it appears in the correct subdirectory of the
classpath.
```

This often means that your computer has two different versions of Java and is confused. If your computer has a copy of Java version 5 or newer, then it cannot use Java version 4 or older (there was a major change with Java version 5 that broke some backwards compatibility). You should uninstall the older version of Java if you know how; otherwise, you should immediately ask either the course staff or consultants for help.

Eclipse Tutorial

We show you how to install and use the IDE (Integrated Development Environment) Eclipse.

Installation Process

Before you install Eclipse (now version 3.6.1), make sure that you have already installed the Java 5 (or 1.5) platform, Standard Edition (Java SE), which consists of the Java Development Kit (JDK) and the Java Runtime Environment (JRE). If not, follow the link on the Java resources on the course website, and follow the instruction for installing Java.

To download Eclipse, visit website http://www.eclipse.org/; click the download button; and choose to download Eclipse SDK 3.6.1. Choose a mirror close to you, and the download window should automatically come up. After the download is finished, you should have a .zip file. Unzip it to the location you want.

You will see a folder named eclipse. Inside the folder, there is an icon that looks like a big blue dot (or Death Star) named eclipse. Double-click it to start Eclipse. You may wish to create a shortcut on the desktop to simplify the launching of Eclipse.

Launch Eclipse

After you launch Eclipse, it will ask you to specify your workspace, which is a folder in which Eclipse stores all your work for this session. After you specify it, press button OK to continue. Then, you will see some kind of introductory page. Click the curved arrow on the right to go to the main working windows.

How to create a new project and a new class

Each Java program you write needs to be contained in its own project, which is a folder that contains every class in it. So, we first need to create a new project. To do that, click `File` and choose `New > Java Project.` (Alternately, click the black downward-pointing triangular arrow next to the rectangle in the top left that looks like a mini-window with a blue menu bar and a diamond or star in the upper right corner of it; and select `Java Project` from there. The little diamond is supposed to indicate something shiny and new.) You'll get a dialog box that asks for the Project name; type it in. In this example, we will put `PandaProject` and then click button `Finish`. We have created Java project `PandaProject` (basically a folder in the workspace location).

We are ready to create a new class. There are several ways to do this. You can choose from the menu bar as we did earlier. Click `File` and choose `New > Class`. Another way is to click the same the black downward-pointing triangular arrow [next to the rectangle in the top left that looks like a mini-window with a blue menu bar and a diamond or star in the upper right corner of it] and choose `Class` instead of `Java Project` … Also, you can right click the name of the project in the `Package Explorer` tab, then choose `New` then `Class`.

The New Java Class window should open. Type in the name of the class —we use `Panda`. At the bottom of the window are several options. One option you may need is method main, which is necessary for running the main program. Then, click button `Finish`. Now you will see your class appear below the project name in the `Package Explorer` window.

How to run a project

Before you try to run a Java program in Eclipse, make sure that you have a method main in one of the classes in the project. It is required for running your program. Again, you can obtain main method by selecting it when you first created the class or by manually typing **`public static void`** `main(String[] args)` as a method header. (Complete the body appropriately, of course).

To run the program, you can click `Run` on the menu bar and choose `Run.` You can also click the icon that is a green circle with a white triangle in the middle. It has a small arrow next to it, so that you can choose `Run.`

If you want to do something fancier, like run the program using some specified arguments, things get a little more complicated. Choose the `Run configurations…` option, which will pull up a tabbed window. However, you need to know only a few major things. On the left side, there is the `Configuration` window, where you can choose to run different types of Java programs. We usually use the `Java Application`. You will see a collection of your classes. Choose the one you want to run (which contains method `main`). Several

tabs will appear on the right. Check at tab `main`. Make sure that the name of the class and project are correct. If not, you can click button `Browse ...` to find the right one. To add arguments, click tab `Arguments`. We are interested in `Program` arguments. Here is where you can put your arguments. For example, if you want the first argument to be `"one"`, the second argument to be `"2"`, and the third one to be `"Cornell"`, put `"one 2 Cornell"` in the `Program` arguments box.

Now click button `Run`. The result of execution will appear in the Console window at the bottom of your main window.

Creating a JUnit class

Creating a `JUnit` class is not as simple as in DrJava. First, right click the class for which you want to add `JUnit` test cases. A window will appear asking if you want to add `"junit.jar"` on the build path. Say Yes. If you don't know what it is, that is fine. The actual window for creating a new `JUnit Test Case` will appear. The name of your class will be set automatically. In this case, we will put `"PandaTester"`. Change the name to whatever you want. You may also want to add a method `main`, which is necessary for running a Java program in Eclipse.

Additional Information

Later, when you develop a large-scale project, you will want to learn more about functionality of Eclipse, such as adding external .jar files or debugging tools. Also, Eclipse comes with a lot of plug-ins, for example, for GUIs, and even a DrJava plugin. You can find many good tutorials on the internet.

Types and type int

A *type* is a set of values together with the (primitive) operations on them. Memorize this definition! The term will be used over and over again when dealing with Java.

Type *integer* is used frequently in mathematics, the sciences, and engineering:

Type *integer*:
values: the set of integers { …, -3, -2, -1, 0, 1, 2, 3, …}
operations: *negation*, or *unary minus* –b
addition b + c,
subtraction b – c,
multiplication b * c,
division /.

Integers have to be stored in memory locations on a computer, which have finite capacity. Therefore, Java, like most programming languages, does not use type *integer*. Instead, it has a type **int**, in which the values are restricted to the range -2^{31} to $2^{31}-1$. This range is chosen because of the way the numbers are stored in today's computers, in a sequence of 32 bits, or 4 bytes. You don't have to know exactly how the values are stored in 4 bytes.

Operations on type int

The operations of type **int** are those of type integer, except that a new one, remainder %, has been added. We will go over these in a minute. But before we do, we mention an important principle decided upon by the designers of Java:

Principle: Each primitive operation of type **int** must yield an **int**.

We use this principle to help answer questions about type **int** and its operations.

We now use DrJava's interactions pane to investigate some properties of type **int**. First, unary minus, addition, subtraction, and multiplication work as you would expect them to.

But what about division? According to the principle that **int** operations must yield **int**s, 6 / 4 cannot yield 1.5. There are two possible choices for the value of 6 / 4: either truncate to 1 or round up to 2. The Java designers chose truncation toward zero, so 6 / 4 = 1 and –6 / 4 = –1.

The percent sign % is used for the remainder operation 6 % 4 is the remainder when 6 is divided by 4. So, 6 % 4 = 2 and 6 % 3 = 0.

Names for the minimum and maximum values of type int

Java has names for the smallest and largest values of type **int**:

Integer.MIN_VALUE and Integer.MAX_VALUE

Later, we will see why it has these values have these strange names, with a period in them. For now, just use them.

Since Integer.MAX_VALUE is the largest value of type **int**, what do you think adding 1 to Integer.MAX_VALUE yields? There are several possibilities, and the designers of Java had to choose one. What do you think it should be? Note that, there is no "wrong" answer. There are reasons for each of several choices, and the designers just chose one of them.

One choice would be to have evaluation of Integer.MAX_VALUE + 1 give a error and halt execution, so that the programmer, or user, knows that some mistake has been made. Another might be to yield Integer.MAX_VALUE itself, and one might even say it should be 0. Java chose another answer; Integer.MAX_VALUE + 1 yields the smallest value of type **int** —the values "wrap around". To know why this value was chosen, one has to know something about arithmetic is implemented in the computer, and we don't go into that here.

The programs you write in this course will exhibit this wrap-around, and you need not worry about this kind of "overflow" now.

Java actually has four different integer types, each with a different range of values, so that programmers can control to some extent the memory used by a program. We have seen type **int**. A value type int uses 32 bits, or 4 bytes.

Values of type **byte** occupy one byte (8 bits) of space.
Values of type **short** occupy 2 bytes (16 bits).
Values of of type **long** occupy 8 bytes (64 bits).

You do not have to remember all these details about the integer types in Java. In fact, in this course, we will be using only type **int**. However, if you ever need information about any of these types, turn to Chapter 6 of the text for this course. Use this chapter often as a reference, whenever you have questions about the primitive types in Java.

The ProgramLive CD has an extensive, hypertexted glossary of definitions, which can be extremely useful in finding out information about Java and programming. For example, here is page 6-1 of ProgramLive. If we click on a blue underlined word, the glossary opens to the corresponding entry for that word. This glossary provides an efficient, effective way for you to explore the meanings of words related to programming.

Type char

Overview

The values of type `char` are the characters that you can type on your keyboard, plus many more.

Literals of type `char` are written with single-quote marks surrounding the character. The last example below is the blank, or space character, which you get by stroking the space bar on your keyboard.

```
'a'        'Z'        '<'        ' '
```

Try typing `' '` into the DrJava interactions pane, and you are in for a surprise. It produces an error message. The problem is that the second single quote is ambiguous —does it indicate the end of the literal or does it represent the single-quote character? To get around this ambiguity, Java and many other languages use the escape character `\` to write characters like the single quote, like this:

```
'\''
```

The table below lists four characters that are written using the escape character. To see others, look at page 225 of Gries/Gries.

character	meaning
\'	the single quote char
\"	the double quote char
\\	the backslash char
\n	the new-line char

You will see how the new-line character is used in the next lecture, on Strings.

Representation of characters: ASCII and Unicode

You don't have to read the information below on the representation of characters and Unicode. However, it will give you a sense of history and help you understand how inclusive type char is with respect to the world's languages.

Representation of characters

In the 1960's (and before, in some places), characters were represented in a 7-bit format known as ASCII (American Standard Code for Information Interchange). This code had representations of small and capital letters, many other characters that you find on your keyboard (e.g. `!@#$%^&-+=<>`) and many non-printable characters, like tabs and line-feeds.

For example, the representation for ’A’ was the integer 65, or, in binary, 1000001. ASCII defines codes for 128 characters, 33 of which are non-printing (like the tab).

But there are many other alphabets besides the one we use in the United States —Russian, Finnish, Greek, Japanese, Persian, and so forth, to say nothing of Chinese. A truly international code would allow characters from all these alphabets.

In 1991, the Unicode Consortium was created to develop a 16-bit code that would include all alphabets. The outcome of their work was the Unicode Standard, which is now in use in many programming languages, including Java. The ultimate reference for Unicode is the website www.unicode.org/. To get a simpler and shorter glimpse of Unicode, turn to lesson page 6-5 of the CD ProgramLive.

Writing unicode representations in Java

In Java, you can write the character 'A' as the character literal ’\u0041’. The ’u’ indicates that the next 4 characters give the hexademical unicode representation of the character in question. To show you how comprehensive the Unicode standard is, below, we give a table of some characters along with their representations as Java char literals, their decimal representations, and the languages to which they belong some of them might not print properly in some browsers. For fun, type the unicode representation into DrJava's interactions pane and see the letter itself appear.

character	Java unicode	decimal representation	language
'A'	'\u0041'	65	Latin, English
'Ճ'	'\u0543'	1347	Armenian
'א'	'\u05D0'	1488	Hebrew
'Ꭼ'	'\u13DC'	5084	Cherokee
'औ'	'\u0914'	2324	Devanagri (Sanskrit)
'グ'	'\u30B0'	12464	Japanese Katakana
'گ'	'\u06B0'	1712	Arabic
ㄘ	'\u3118'	12568	Chinese Bopomofo

Type double

Overview

The values of type `double` are approximations to the real numbers. Examples are

```
1.8          3.0          0.000005
```

We do not make heavy use of type `double` in this course, and we present only a basic outline here. For more information see section 1.1.2 (page 18) in Gries/Gries. Also, use Gries/Gries, section 6.4, pp. 221–224 and lesson pages 6–3 and 6–4 on the Program live CD as references for type double.

Values of type `double` are approximations to the real numbers because real numbers have an infinite number of decimal places while each `double` number occupies a fixed number of bits. There are an infinite number of real values and only a finite number of `double` values. For example,

> `0.1111111111111111111111111111111` in the DrJava interactions pane evaluates to `0.1111111111111111`

And, amazingly,

> `1.0/9.0` in the interactions pane evaluates to `0.1111111111111111` but
> `0.1/0.9` in the interactions pane evaluates to `0.11111111111111112`.

In mathematics, we use scientific notation, like $3.1*10^7$, which stands for 3.1 multiplied by 10,000,000. In Java, this number would be written as `3.1E7`. In this notation, the number to the left of `E` is called the mantissa and the number to the right is called the exponent. The mantissa is truncated to about 16–17 decimal digits.

Obtain the maximum and minimum positive values using:

```
Double.MIN_VALUE: 4.9E-324
Double.MAX_VALUE: 1.7976931348623157E308
```

The following table gives the operations of type `double`. They perform as expected, except that "round-off" errors will occur because of the finite nature of `double` numbers.

Operation	Name	Example
unary `-`	negation	`-3.8`
unary `+`	no-op	`+7.13E11`
`+`	addition	`1.3E7 + 2.4E8`
-	subtraction	`3.14 - 2.73`
*	multiplication	`8.0 * 2.5`
/	division	`1.0 / 9.0`

Operation	Name	Example
`%`	remainder	`5.1 % 2.0`

If the operands of unary minus, `-`, `+`, `*`, `/`, `%` are of type `int`, the operation is an `int` operation. If at least one operand is `double`, the operation is a `double` operation, and, if necessary, the other operation is first converted to type `double`.

Type boolean

In some programming languages, for example Matlab and C, integers are used to represent the logical values **`true`** and **`false`**. Generally, 0 is used for **`false`**, and any other integer can be used for **`true`**.

Java handles boolean values differently. There is a (primitive) type **boolean**, whose values are **`true`** and **`false`** (that's it). This type has five operations whose operands are booleans:

Type **`boolean`**
Values: **`true`**, **`false`**
Operations: `!` (*not*), `&&` (*and*, or *conjunction*), `||` (*or*, or *disjunction*)
`==` (equality, or equivalence), `!=` (inequality, or inequivalence)

Here is a table that defines the five operations.

b	c	!b	b && c	b || c	b == c	b != c
false	false	true	false	false	true	false
false	true	true	false	true	false	true
true	false	false	false	true	false	true
true	true	false	true	true	true	false

We evaluate a few expressions in the interactions pane and discuss the operations.

1. !, which is read "not", is unary logical negation **`!false`** is **`true`**, and **`!true`** is **`false`**.

2. `&&` is read *and* because `b && c` is true iff both `b` and `c` are true.

3. `||` is read *or* because `b || c` is true iff either `b` or `c` (or both) is true.

4. `==` is used for equality: `b == c` is true iff `b` and `c` have the same value.

5. `!=` is used for inequality: `b != c` is true iff `b` and `c` have different values.

Relations

Six relations operate on the numeric types to yield boolean values.

`b == c, b != c, b < c, b <= c, b > c, b >= c`

You have probably seen these relations in other programming languages, so we don't go into full details here. The only strange point is that == is used for equality, and not =. Here are examples.

`5 < 6` is **`true`**
`5 >= 6` is **`false`**
`5 < true` is illegal because one operands is an **`int`** and the other a **`boolean`**.

These relational operators work for all the number types —**int**, **double**, **char**, etc. For example, we can test whether 5 < 6.2 is true, or whether 6.0 == 7 is true. If the two operands are not of the same type, one is converted to the other type so that the operation can be carried out. More on such conversions later.

Short circuit evaluation

Evaluation of

`5/0 == 3 && false`

results in an error, because of the division by 0. This is to be expected. But evaluation of the same expression with the operands reversed,

`false && 5/0 == 3`

does not produce an error message —it yields the value **false**. This is because evaluation of `&&` is done in *short-circuit* mode: as soon as the answer is known, evaluation stops. Since **`false`** `&&` `b` is always false, no matter what b is, there is no need to evaluate b.

Another way to look at the evaluation of b && c is to say that it is equivalent to an if-expression **if** b **then** c **else false**, which can actually be written in Java using the expression

`b ? c : false` // equivalent to `b && c`

You will see this conditional expression later. Get used to it; it is useful.

In the same way, **`true || c`** is **`true`** no matter what the value of `c` is, so `c` is not evaluated in this case. The expression `b || c` is equivalent to

`if` `b` **`then true else`** `c`, or the Java expression `b ? true : c`

As you will see in several assignments, short-circuit evaluation is a useful tool in writing boolean expressions.

Type String

Type `String` is different from primitive types **`int`** and **`double`**. `String` is a *class* type. We explain what this means later, after we have explained what a class is.

Each value of type `String` is a sequence of characters. We write a value by enclosing the sequence of characters within double quotes `"`. You can see in the interactions pane of DrJava that these literals evaluate to themselves. The second example contains a capital letter and three occurrences of the space, or blank character, which is a character. The last example shows the *empty string*, the string containing no characters.

```
"first"    "The price is $25.00."    "5 <= 20"    ""
```

Use a new-line character `"\n"` to indicate a jump to a new line. For example, notice how evaluation of `"123\n4"` appears on two lines. The new-line character begins with the so-called escape character, the back-slash. Use the escape character to write the backslash itself, as well as the double quote character. For example, note how the String literal `"1\\2\"3"` evaluates to a String containing can contain the characters 1. \, 2, ", and 3. Look in the text, Chapter 6, to find other characters that are written using the escape character.

Catenation (String +)

The *one* operation of type `String`: catenation (or concatenation), is used to append one string to another. The result of the expression

```
"Store this" + " in s."
```

is the value

```
"Store this in s."
```

Here's a useful rule:

> If one operand of + is a `String`, the other operand is converted to a `String` and catenation is performed.

We look at some examples of the use of this rule. First, watch how the constant 1 is changed into a `String`. Second, watch how two numbers are first added, because of the parentheses, and the result is changed to a `String` so that catenation can be performed. If the parentheses are removed, the + operations are carried out left to right, so that both +'s are catenations:

```
"three" + 1       // yields  "three1"
"three" + (1+2)   // yields  "three3"
"three" + 1 + 2   // yields  "three12"
```

Here's a useful trick: use catenation with an empty string to turn the value of any expression into a `String`.

`"" + (945 - 3/2.0)` yields `"943.5"`

Useful functions

Type `String` provides a number of functions. We explain four of them.

length. For a `String s`, `s.length()` yields the number of characters in `s`. This rather funny notation, with the period in it, will make sense after we introduce classes.

```
"love".length()  // yields 4
```

You can obtain the length of any `String` expression, and not just of a literal. Here, the catenation is first performed to produce a `String` value, and then its length is obtained.

```
("non" + "violence").length()
```

The position or index of a character

Two functions are used to get at subparts of a `String`, To understand these, you have to know that the first character of a `String` is at position 0, the second is at position 1, ..., and so on. So, numbering of the characters begins at 0 and not at 1! The position is more frequently called the *index* of the character.

charAt. For a `String s`, `s.charAt(i)` gives the character at position `i` of `s`. Here are examples.

```
"Truth".charAt(0)    // yields 'T'
"Truth".charAt(4)    // yields 'h'
```

For `t.charAt(i)` to be legal, `i` must be a valid position number, or index. If not, execution aborts with an error message. For example, `"T".charAt(2)` causes a `StringIndexOutOfBoundsException`, giving the invalid index. So be careful. Whenever you write a function call `charAt(i)`, make sure `i` is in the appropriate range.

substring. Function substring extracts a substring of a string. For example,

```
"0123456".substring(1,4);
```

extracts the string consisting of the characters at positions 1, 2, and 3, but not 4. The first argument says where to start. The second argument gives the index of the character *after* the last one to be extracted.

If you leave off the second argument, the result contains every character from the first argument to the end of the string:"

```
"harmony".substring(2);   // yields "rmony"
```

A substring call with equal arguments yields the empty string `""`.

`"harmony".substring(2,2)` is `""`

You can even ask for an empty substring of the empty string:

`"".substring(0,0)` yields `""`.

equality. Please don't use `s1 == s2` to test whether strings `s1` and `s2` are the same, because it won't work the way you think —because String is a class type. Again, you'll have to wait until we discuss classes to understand this. Here's an example to show that `==` doesn't work:

```
"abc" == "ab" + "c"       // is false
```

The proper way to test equality of strings is to use function `equals`:

```
"abc".equals("ab" + "c")  // is true
```

Function calls

You have seen *function calls* since high school, like

```
sqrt(25)      // square root of 25. Its value is 5.0.
abs(-7)       // absolute value of -7. Its value is 7.
cos(0)        // cosine of angle 0. It value is 1.0.
min(5, 7)     // minimum of 5 and 7. Its value is 5.
```

You will learn to write functions in Java later. Here we just look at how to write *calls* on functions.

A function call has the form

<function-name> (*<arguments>*)

where *<arguments>* is a list of expressions separated by commas. The first three function calls you see here have one argument. The last function call has two arguments —the expressions `5` and `7`— and they are separated by a comma. You have even seen a function with no arguments: function `length` (remember the lecture on type `String`).

Please remember the term *argument* of a function call. You may see it called the *actual parameter*. We do not use this terminology because it is confusing, as you will see when we discuss writing functions themselves.

A function call is an expression, and you can put one anywhere an expression could be placed, as you can see in this expression:

```
1 + min(5, 7) * 2
```

Secondly, since a function call is an expression, it can be *evaluated* to *yield a result*. The value the expression `min(5, 7)` is `5`, so the value of the expression given above is `11`.

Since Java is strongly typed, arguments of function calls have to have certain types. For example, you can't ask for the square root of a boolean value. And the value of a function call also has a specific, known type.

Java has thousands of built-in functions for you to use. Here, we tell you about a few of the mathematical ones. Their names begin with "`Math.`" —for reasons that will become clear later.

Function `Math.sqrt` yields the square root of its argument. Its argument is a **`double`** value —if it is a narrower type, like **`int`**, the argument will be promoted to a **`double`**. And it yields a **`double`** result.

```
Math.sqrt(double)        // Yields a double.
    Math.sqrt(25.0)      // The square root of 25.0. Its value is 5.0.
    Math.sqrt(5.0)       // The square root of 5.0. Its value is 2.23606797749979.
    Math.sqrt(25)        // The square root of 25.0. Its value is 5.0.
    Math.sqrt(5)         // The square root of 5.0. Its value is 2.23606797749979.
```

Function `Math.abs` yields the absolute value of its **`int`** argument —found simply changing the sign of the argument, if necessary, so that the answer is positive.

```
Math.abs(int)            // Yields an int.
    Math.abs(5)          // The absolute value of 5. Its value is 5.
    Math.abs(-5)         // The absolute value of -5. Its value is 5.
```

Actually, there are several absolute functions, depending on the argument type. For example,

```
Math.abs(double)         // Yields a double.
    Math.abs(5.0)        // The absolute value of 5.0. Its value is 5.0.
    Math.abs(-5.0)       // The absolute value of -5.0. Its value is 5.0.
```

Function `Math.min` yields the minimum of its two **`int`** arguments. And, you can see by testing in the interactions pane that there are two min methods, one for **`int`** arguments and another for **`double`** arguments. Note that if one argument is a **`double`** and the other is an **`int`**, the **`int`** value will be automatically promoted to type **`double`.**

```
Math.min(int, int)       // Yields an int.
    Math.min(5, 7)       // minimum of 5 and 7. Its value is 5.
```

```
    Math.min(5, -7)        // minimum of 5 and -7. Its value is -7.

Math.min(double, double)// Yields a double.
    Math.min(5.0, 7.0)     // minimum of 5.0 and 7.0. Its value is 5.0.
    Math.min(5.0, -7)      // minimum of 5.0 and -7. Its value is -7.0.
```

Three useful functions are

`Math.round(double)`, `Math.floor(double)`, `Math.ceil(double)`.

Math.round rounds its argument to the nearest integer:

```
    Math.round(5.6)        // Its value is the int 6.
    Math.round(5.4)        // Its value is the int 5.
```

`Math.floor` and `Math.ceil` are the floor and ceiling function. Think of one integer, say 5, as being the floor, the next integer, say 6, being the ceiling, and all the real numbers between 5 and 6 being between the floor and ceiling.

A more mathematical definition is that

`floor(x)` is the largest integer that is at most `x`.
`ceil(x)` is the smallest integer that is at least `x`.

Note that the result, although an integer, is of type **`double`**.

```
Math.floor(5.6)     // Its value is the double 5.0.
Math.floor(5.4)     // Its value is the double 5.0.
Math.ceil(5.6)      // Its value is the double 6.0.
Math.ceil(5.4)      // Its value is the double 6.0.
```

There are many more functions like this. We just want to give you an idea of what is available. You can find more of them listed in a table on page 24 of Gries/Gries, and later, we will show you how to access the specifications of all the functions that begin with "`Math.`". Also, we wanted to acquaint you with our terminology for dealing with functions.

Precedences of operators

You know that multiplication * takes precedence over addition +. That is, the expression 5 + 4 * 3 is evaluated as if it were parenthesized like this: 5 + (4 * 3).

Mathematics has conventions for precedences of operators in order to reduce the number of parentheses required in writing complex expressions. Some of these conventions are standard throughout the world, such as * over +. Others are not.

Below is a table of precedences for Java operators. Refer to this table, or the one on p. 227 of Gries/Gries, if you forget the precedences.

Table of operator precedences

ORDER	OPERATORS	EXAMPLES
Highest	Unary ops:	+ − ++ — !
	Binary arithmetic ops.	* / %
	Binary arithmetic ops.	+ −
	Arithmetic relations:	< > <= >=
	Equality relations:	== !=
	Logical and:	&&
Lowest	Logical or:	||

For example, the expression

```
(n != 0) && (10/n > 2)
```

can be easily rewritten as

```
n != 0  &&  10/n > 2
```

because relational operators have precedence over &&.

Some Guidelines

Keep redundant parentheses to a minimum (but don't sacrifice clarity)

Your goal in writing expressions should be to make them a clear and simple-looking as possible, and this can often be done by eliminating redundant expressions —but you can help the reader by putting more space around operators with less precedence. For example, the two assignment statements shown below have the same meaning, but the second is easier to read:

```
b= ((x<=y)||(x<=z));
b=  x <= y  ||  y <= z;
```

Keep redundant parentheses to a minimum. For example, they are not needed in a return statement within a function body (you will learn about the return statement later). The following two statements are equivalent, but the second is easier to read:

```
return (x >= -10 && x <= 10);
return x >= -10  &&  x <= 10;
```

Use parenthesis with AND and OR for clarity

Java gives && precedence over ||, so that b || c && d is equivalent to b || (c && d). This convention is non-standard, and controversial for some. Therefore, when writing sequences of &&s and ||s, we suggest always putting in parentheses to make the meaning clear.

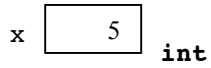
This is a case where the suggestion to eliminate redundant parentheses is overruled in favor of removing confusion and providing clarity.

Variables and declarations

A *variable* is a name with an associated value. Throughout this course, we draw a variable as a named box with the value in the box. For example, here is a variable `x` associated with the value 5. Sometimes, we draw a line instead of a box, as shown to the right below:

Java is *strongly typed*, meaning that it has properties like the following.

1. *Each variable must have a type*. When we want to make the type of a variable explicit, we put the type to the right of the box.



2. *Only values of that type may be associated with the variable*. So, variable x, declared above, cannot contain a boolean value **true** or **false**.

3. Each expression has a type, which depends on the types of its operands, and

4. An operation can be applied only to operands of the appropriate type.

For example, an **`int`** variable may contain only an **`int`** value. A **`boolean`** variable may contain only a **`boolean`** value, and so on. Also, operation `b && c` is legal only if `b` and `c` are of type `boolean`.

Matlab is not strongly typed —one second, a variable may contain a complex number, the next second, an array of characters. The programming language C is very weakly typed —nothing prevents one from treating a string (character array) like "whatever" as a struct, an array of 32-bit integers, or anything else.

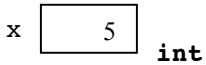
A later short essay tells you the advantages and disadvantages of strong typing.

Because of the strong typing principal, each variable must be declared before it is used. A variable declaration has the basic form

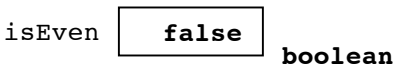
<type> <variable-name>

For example, here are two declarations and the corresponding variables, where we have annotated the variable itself with its type:

```
int x
```




```
boolean isEven
```



We make two points about declarations.

First, depending on where the declaration appears in a program, it may be followed by a semicolon or a comma, and it may also be preceded by an "access modifier", which indicates what part of the program can reference it, and other things. But the two basic parts —type and variable name— will always be required.

Second, the declaration is *not* a statement to be executed. It is simply an announcement that a variable is needed in a program. A variable declaration is *never* executed.

Declarations in the interactions pane

Let's put a declaration in the interactions pane and see what the value of the variable is:

```
int k;
k              // the initial value of k is 0
```

Note that if we try to declare k again, Java complains. Each variable can be declared only once.

Java conventions for variable names

Here are rules for identifiers —names of variables. They are:

(1) A variable contains only letters, numbers, and the characters `'_'` and `'$'`.

(2) The identifier does not start with a number.

But most Java programmers follow the convention that

(1) Identifiers do not contain `'$'` —this char is used at times by the compiler.

(2) The first letter of a variable name is not a capital.

(3) If the name consists of a series of words, the first letter of each word is capitalized, and

Here are examples of good variable names:

```
apple    numberOf7s    isEven    numberOfChildren
```

The length of variables names is always a cause for argument. Make them too short, and you don't know what they mean. Make them too long, and the program looks cluttered and unwieldy.

Later, we give conventions for variable names depending on where they are declared, keeping them short if their use is close to the declarations and making them longer if their use if far from their declarations.

The assignment statement

The assignment statement is used to store a value in a variable. As in most programming languages these days, the assignment statement has the form:

<*variable*>= <*expression*>;

For example, once we have an **int** variable `j`, we can assign it the value of expression `4 + 6`:

```
int j;
j=  4+6;
```

As a convention, we always place a blank after the = sign but *not* before it. You don't have to follow this convention. Our reasons for it are explained on p. 27 of Gries/Gries.

Once we have variables with values, we can use those variables in expressions. For example, we can evaluate the expression consisting simply of `j`, or the expression `2*j`, and we can store the value of expression `j+1` in another variable k:

```
j
2*j
int k;
k= j + 1;
k
```

You must memorize how the assignment statement is executed, or carried out. If asked, you should say:

Evaluate the <*expression*> and store its value in the <*variable*>.

Please memorize this definition of how to execute the assignment statement. In order to be sure that you understand it, we execute a series of assignments, showing how the variables change.

Here's variables `j` and `k`, with the values computed by what we have done so far. We now execute a sequence of three assignments. Add 2 to `j` and store the result in `j`. subtract `k` from `j` and store the result in `k`, and store 0 is in `j`.

```
j= j + 2;
k= j - k;
j= 0;
```

As we carry out the assignments, we change the values of the variables. We do *not* draw the variables again. There is only one variable `j`, and its value is changed whenever `j` is assigned a new value.

The initializing declaration

We can abbreviate a declaration of `c` followed by an assignment of 25 to it, using an *initializing declaration*:

```
int c= 25;
```

Actually, any expression may be used —the expression need not be a constant.

It is important to realize that this is simply a combination of a declaration and an assignment. Writing two such initializing declarations for the same variable will not work because only one declaration per variable is allowed.

```
int m= c+1;
int m= 45;   // illegal because m has already been declared
```

The types of variable and expression must match

In a Java assignment, the types of the variable and expression must match. For example, if one is a **boolean**, the other must be also, and if one is a `String`, the other must be a `String`. This is a consequence of the strong typing principle.

For numeric types, there is a bit more leeway. You know that there are types **byte**, **short**, **int**, and **long** which have increasingly larger sets of values, and there are two floating point, or real-number, types, **float** and **double**. These move from the so-called narrowest type **byte** to the widest type, **double**.

The rule for an assignment of an expression that is a number is that the type of the variable has to be at least as wide as the type of the expression.

For example, if we have, if we have a **`byte`** variable `b` and an **`int`** variable `i`, both of which contain 0, it is legal to assign `b` to `i` but illegal to assign `i` to `b`.

```
byte b= 0;
int i= 0;
i= b;
b= i;    // illegal
```

The reason for the rule should be clear. Assigning a wider-type value to a narrower-type variable may lose information or result in overflow of some sort. For example, how could 6000 be stored in a byte variable?

You might think that Java would allow an assignment of an **`int`** to a **`byte`** but would complain at runtime if the **`int`** value were too big. However, this would violate the strong typing principle, as designed in Java.

Casts & Mixing Primitive Types

Converting one type of data into another must follow the rules of *casting*. If a conversion results in the loss of precision, as in an `int`value converted to a `short`, then the compiler will issue an error message unless an explicit cast is made.

To convert type `AA` data into type `BB` data, put the type `BB` name in parentheses in front of the type `A`A data:

```
AA a = aData;
BB b = (BB)a; // cast type AA to type BB
```

For example, to convert integer data to floating point:

```
int i=0;
float f;
f=(float)i; // Cast int as float
```

Expressions can promote to a wider type without an explicit cast:

```
 int i=1;
long j=3L; // Literals are int types so require L suffix
j=i;             // OK
```

However, you can not assign a value to a more narrow type without an explicit cast**:**

```
i=j;             // Error in assigning long to int
i=(int)j;   // OK
```

So a data type with lower precision (fewer bits) can be converted to a type of higher precision without explicit casting. To convert a higher precision type to a lower precision, however, an explicit cast is required or the compiler will flag an error.

Note that when you cast a value of a wider type down to a more narrow type, such as an `int` value to a `byte` variable, the upper bytes will be truncated. That is, the lowest order byte in the `int` value will be copied to the `byte` value.

Primitive Type Conversion Table

Below is a table that indicates to which of the other primitive types you can cast a given primitive data type. The symbol **C** indicates that an explicit cast is required since the precision is decreasing. The symbol **A** indicates that the precision is increasing so an automatic cast occurs without the need for an explicit cast. **N** indicates that the conversion is not allowed.

	int	long	float	double	char	byte	short	boolean
int	-	A	A*	A	C	C	C	N
long	C	-	A*	A*	C	C	C	N

float	C	C	-	A	C	C	C	N
double	C	C	C	-	C	C	C	N
char	A	A	A	A	-	C	C	N
byte	A	A	A	A	C	-	A	N
short	A	A	A	A	C	C	-	N
boolean	N	N	N	N	N	N	N	-

The * asterisk indicates that the least significant digits may be lost in the conversion even though the target type allows for bigger numbers. For example, a large value in an `int` type value that uses all 32 bits will lose some of the lower bits when converted to `float` since the exponent uses 8 bits of the 32 provided for `float` values.

Mixed Types in an Expression

If an expression holds a **mix** of types, the lower precision or narrower value operand is converted to a higher precision or wider type. This result then must be cast if it goes to a lower precision type:

```
float x,y=3;
int j,i=3;
x= i*y;             // OK since i will be promoted to float
j= i*y;             // Error since result is a float value
j= (int)(i*y) // OK
```

The process of converting a value to a wider or higher precision integer or floating point type is called "numeric promotion". The Java VM specification states the following rules for promotion in an expression of two operands, as in `x+i`:

- If either operand is of type `double`, the other is converted to `double`.
- Otherwise, if either operand is of type `float`, the other is converted to `float`.
- Otherwise, if either operand is of type `long`, the other is converted to `long`.
- Otherwise, both operands are converted to type `int`.

The Conditional Expression

The purpose of the conditional expression is to select one of two expressions depending on a third, boolean, expression. The format for the conditional expression is

<boolean-expression> ? <expression-1> : <expression-2>

If the value of the <boolean-expression> is true, the value of the conditional expression is the value of <expression-1>; if it is false, the value of the conditional expression is the value of <expression-2>.

An important point is that only one of <expression-1> and <expression-2> is evaluated when evaluating the conditional expression. For example, the divide-by-zero in the following expression causes no error because it is not evaluated:

```
true ? 7 : 7/0
```

Expressions <expression-1> and <expression-2> may be any expressions, as long as they have the same type (if one is wider than the other, the narrower one will automatically be promoted to the wider type).

The conditional expression is known as a *ternary operation*, because it has three operands.

Examples of Use

You can use a conditional expression anywhere you could use a more conventional expression. The following example uses the conditional expression to set c to the larger of a and b:

```
c= a > b ? a : b;
```

Be careful. The conditional operators ? and : have the lowest priority of all operators. It is helpful to use parenthesizes around the conditional expression in complicated situations.

Here's an example that creates a string that indicates which is the larger of a and b:

```
"The larger of a and b is " + (a > b ? "a: " + a : "b: " + b)
```

Try writing a few conditional expressions in the interactions pane in order to get familiar with them. For example, write an expression that finds the smallest of three values b, c, and d.

Use the Conditional Expression with Care

Used indiscriminately, the conditional expression can be hard to read. However, in some cases, its use can result in shorter and simpler code. For example, consider setting a variable s to the catenation of two strings: (1) either "Fahrenheit" or "Centigrade",

depending on whether b is true or false, and (2) ″ hot″ or ″ cold″, depending whether c is positive or not. We write:

```
s= (b ? ″Fahrenheit″: ″Centigrade″) + (c > 0 ? ″ hot″ : ″ cold″);
```

In order to write this assignment without using the conditional expression, you would need a sequence of statements and would probably use at least two conditional statements if (...) ... else ....

Strong versus Weak Typing

Safety

One property that is sometimes looked for in a programming language is *safety*. There is some confusion as to what safety means, but here is a definition used in the year 2000 by an ad hoc committee that recommended that the Advanced Placement (AP) test in programming be based on the programming language Java:

> *Safety*. Any attempt to misinterpret data is caught at compile time or generates a well-specified error at runtime.

C and C++ are not safe. An array index operation `b[k]` in these languages may be allowed even though `k` is outside the range of the array. There is also no built-in protection against using pointer manipulation to access almost any part of memory. "Buffer overflow" —akin to writing outside the boundary of an array— is exploited by many computer viruses. Of course, a programmer can explicitly check for overflow, but the language does not require it.

In the hands of a professional, the "unsafe" features can be used to advantage. But for most programmers, a language that provides safety can be of immense value. Especially for beginning programmers, safety helps.

Strong typing

Much (but not all) of safety is enforced by *strong typing* —this term, too, is not well defined, so let's explain what we mean by it here. First, the type of every variable and every expression is a syntactic property —known by looking at the program, without resorting to executing it. Second, a variable may be used only in ways that respect its type —for example, it is impossible to perform double operations on an int value (the int value must first be cast to double format), or to use a boolean value as an integer. Thus, syntactically, it is impossible to have an operation disregard the types of its operands, to interpret a value as something that it is not.

Runtime safety features

The strong typing of Java is backed up by strong guarantees on runtime behavior, the goal being to ensure that no value can be interpreted as something that it is not. With a String value, it is impossible to index an character of the String that does not exist. For example `s.charAt(i)`, will cause an exception if i is not in the range `0..s.length()-1`. Similarly, array subscripts must be in the range of the array. Also, it is impossible to do "pointer arithmetic", as one can do in C and C++. In Java, the pointers are what we call the "names" of objects, and there are no operations on these names.

Benefits of safety and strong typing

Some errors arise from lack of understanding; others are logical errors caused by inadequate thinking and design; and some are simply typos. In all cases, finding errors early, at compile time, can save immense amounts of time. Safety and strong typing make possible the early detection of many errors.

The beginning programmer will benefit from safety and strong typing more than the professional programmer, simply because the beginner tends to make more of each type of error. But the professional programmer will also gain from a safe and strongly type language. When a program consists of hundreds of files, it is too easy to get confused and make mistakes, and the more checks we have on our programs, the better off we are.

Finally, with a language like Java, whose programs can be accessed on web pages (by writing *applets*), safety and strong typing aid the user as well, for these properties make it more difficult for malicious people to hack into systems through such applets.

Folders & file drawers — objects and classes

In a dentist office, you are likely to see file drawers filled with manila folders. The file drawer named *Patient* contains one manila folder for each patient, with each manila folder in the file drawer containing the same kind of information. Here, we see that manila folder a0 contains Jack Smith's name, address, and the amount he owes. In a real dentist office, the folder would contain much more information.

The file drawer is like a *class* in object-oriented programming. And the manila folders in the file drawer are called *objects*, or *instances* of the class. Each manila folder, or object, has a name on its tab, which identifies it.

Drawing objects

Throughout this course, we will need to draw manila folders, or objects, in order to explain various concepts, and you may have to draw such objects, too, when you are trying to find a bug in a program or to explain something to another person. For efficient communication, we must all use the same form of an object. Here it is:

1. It looks like a front view of a manila folder.

2. In the upper right, we draw a box that contains the name of the class —the file drawer in which it appears.

3. Its tab contains a name, which is different from any other name on a tab. When drawing your own folder, it doesn't matter what name you write, as long as it is different from all other such names.

4. The information in the folder is drawn in a series of *fields* —these are actually variables! The order in which these variables are placed does not matter. Here's a term for you to learn: variables placed in an object like this are called *fields*, or *instance variables*. Memorize that name.

The name on the tab

When you draw an object, put any name on it that you like, as long as it is different from all other such names. In the same way, when an object is created during execution of a Java program on a computer, some arbitrary name will be created and placed on the tab of the folder. You have no control —and don't need control— of the name. It is like the address on your house, which is given to you by the local authorities; you have no control over it and you cannot change it.

Methods in a manila folder (or object)

In dentist office, different people will perform different functions on a folder. The secretary will create the folder and enter the person's name, address, and so forth. The accountant will change the amount owed when the person pays a bill. The dentist himself will make notations within the folder.

To mimic this process, we put into the folder itself the sets of instructions that each person follows for each task. These sets of instructions are called *methods* in OO programming. In a dentist's office, putting detailed instructions into the folder would have the advantage that anyone could perform any task on the folder; just follow the instructions.

We work with two kinds of method here.

One kind of method is the *function*, which yields a value. Here, function getName will yield the name of the person whose folder this is: Jack Smith.

A second kind of method is the *procedure*, which performs some task but does not yield a value. For example, procedure deposit could be given the amount of money being deposited, a **double** value, and it would change field owes accordingly.

Later, we will show you how to write methods in Java. They are much like functions or procedures or subroutines in other programming languages. For now, it is just necessary for you to know that methods —functions and procedures— appear in each and every manila folder of the class. This is one of the important structural features of OO programming.

Types and referencing components of an object

A class name like Patient is also considered to be a type. Its values are the names of manila folders, or objects, of that class. The type has no operations, because one shouldn't be able to operate on these names, much like you can't change the address of your house or apartment.

Since Patient is a type, just as one can declare variables of type **int**, one can declare variables of type Patient.

Calling methods

Suppose variable p contains the name a0 of an object. In this situation, in Java, one can call method getName of the object using (1) the name of the variable, (2) a period, (3) the function name, and (4) parentheses:

```
p.getName()
```

For example, we could use this call in an assignment statement to store the name "Jack Smith" in a variable s:

```
S=  p.getName();
```

In the same way, we can call procedure deposit using a statement that consists of (1) the name of the variable, (2) a period, (3) the name of the procedure, (4) parentheses with an argument between them —the amount to be deposited—, and (5) a semicolon:

```
p.deposit(250.00);
```

A statement like this one is a command to do something, and its execution in this situation will change the amount that is owed to 0.

In summary, to write a call on a method of an object whose name is in a variable p,

(1) use the name of the name of the variable followed by a period, and then

(2) a call that is typical in most programming languages, consisting of the method name followed by arguments (if any) in parentheses.

(3) If the method is a procedure, so that the call is a statement, a semicolon is also needed.

We will go over method calls in more detail later.

Fields can be referenced in much the same way.

```
p.name
```

On your screen, you see two examples of this, one assigning p.name to a String variable s and the other subtracting $250.00 from field p.owes.

However, the standard practice in OO programming is to make most fields "private" —we'll see how this is done later— so they cannot be accessed from outside the class. This is standard software engineering practice. The reasons for it will be discussed at the appropriate place, later on.

Value null

On your screen is a declaration of a variable of type Patient. Initially, variable p does not contain the name of an object because none has been assigned to it. Instead, p contains the value **null**, a special value that means the absence of an object name.

In this situation, if the program attempts to evaluate the expression

```
p.getName()
```

a "NullPointerException" will occur and execution of the program will abort. You can't reference a component (a method or a field) of an object that doesn't exist.

Creating an object —the new-expression

Finally, we show you how to create a new object, using a new-expression. Evaluation of the expression

```
new Patient()
```

creates an object and yields as its value the name of the new object. If Java is evaluating the new-expression, it creates an arbitrary name for the object. When *you* evaluate the new-expression and draw a new object, choose any name you want. Here, we have chosen the name a0.

It is important for you to understand that the new expression produces a value —the name of the newly created object. We can say that the new-expression is evaluated in a two-step process, which will be expanded on later:

Step 1. Create a new object of class Patient (giving it a name in its tab)

Step 2. Yield as value of the expression the name on the tab.

Now, we need to store the value of this expression in variable p. Execution of this statement therefore changes the value of p to a0.

Summary

You have seen several new items in this little lecture:

1. How to call a method or reference a field of an object whose name is in a variable p.
2. The meaning of value **null**.
3. The new-expression, whose evaluation creates (draws) a new manila folder or object and has as its value the name of the new object.

In this little lecture and the previous one, you saw powerpoint slides, making the lectures just a bit abstract. In the next lecture, you will see this made concrete as we use DrJava to illustrate what we have discussed thus far.

Objects of class JFrame

Thus far, you have seen powerpoint slides that showed what an object is, how to call methods of an object, how to reference fields of an object, and how to create a new object using a new-expression. We now make these concepts more concrete using DrJava. You will see the creation of new objects, execution of procedure calls, and evaluation of function calls.

In Java, an object of class `javax.swing.JFrame` is associated with a window on your monitor. We will show you later how the rather long name `javax.swing.JFrame` can be abbreviated to simply `JFrame`. Class `JFrame` comes with every implementation of Java.

The upper right pane of DrJava, give a list of the methods that are in each object of class `JFrame`. What they do is indicated in the method names.

We begin by creating a new object of class `JFrame`, using a new-expression, and assigning it to a variable `x`:

```
x= new javax.swing.JFrame();
```

The upper right pane in DrJava shows variable `x` and the new object.

Calling procedures

An object of class `JFrame` has been created, but we can't see the window that is associated with it. So, we execute method

```
x.show();
```

There is the window, in the upper left corner of the screen! Calling method show of object `x` (or, rather, the object whose name is in `x`), caused the window to appear. Let's drag the window down a bit and make it bigger.

To show you the effect of other method calls, we hide the window and then show it again.

```
x.hide();
x.show();
```

We call one more procedure, to set the title of the window. The desired title is given as the argument of the procedure call:

There, the title now appears in the title bar of the window! Do you see how execution of a call of a procedure in an object has an immediate effect?

Calling functions

We call some functions. Methods `getX()` and `getY()` yield the position, in pixels of the upper-left pixel of the window.

```
x.getX()
x.getY()
```

Let's move the `JFrame` over to the left edge of the window and call `getX` again, so you can see that the value of the function call depends on the position of the window:

```
x.getX()
```

Functions `getWidth` and `getHeight` yields the width and height of the window, in pixels. We leave you to write calls on them later on.

Squaring the window

Procedure `setSize` changes the size of the window. The width becomes the first argument and the height becomes the second argument. Let's use `setSize` to square the window by making the width equal to the height. We first write the call, with no arguments, and then fill in the arguments. The first argument should be changed to the height, which is `x.getHeight()`, and the second argument should also be the height. There we are! Look how the window changed shape!

Objects of class JFrame

Each object has its own methods

We create a second object and store its name in a variable y and show the new window:

```
y=  new javax.swing.JFrame();
y.show();
```

There it is!

We now have *two* objects, which are associated with different windows. We can show you that each object has its own methods by setting the title of the second window to something other than the title in the first:

```
y.setTitle("truth");
```

See? The titles are different. We can also set the first title again:

```
x.setTitle(("love");
```

The point should be clear. Each object has its own methods, which manipulate the fields and other properties of that object in some way.

Aliasing

We make one more point. The upper right pane contains a picture of the two variables `x` and `y` and the two objects. Suppose we write an assignment to store `x` in `y`:

```
y= x;
```

How is this executed? According to the definition of the assignment, the value of the expression on the right, which is `a1`, is stored in variable `y`, so `y` also contains the name `a1`. Therefore, two variables contain the same object name. This is called *aliasing*. Aliasing happens often in object-oriented programming

We evaluate both `x.getTitle()` and `y.getTitle()` to show you that they contain the same value, since x and y contain the same name.

Object `a6` can no longer be referenced, because no variable contains its name. If you were executing this by yourself, drawing objects on a piece of paper, you could erase object `a6` because it can never be used again.

Packages

In the previous lecture, we used the class name `javax.swing.JFrame`. In this little lecture, we show how it can be abbreviated as `JFrame`.

In Java, a *package* is a collection of classes that reside in the same directory on your hard drive. In our file-cabinet analogy, think of a package as a room that contains file cabinets with one drawer for each class.

The package is Java's way of organizing a large collection of classes —Java version 6 comes with over 3775 classes, organized into about 200 packages. Here are four of the packages.

package `java.io`	classes dealing with input/output
package `java.net`	classes dealing with the internet
package `java.awt`	classes dealing with GUIs
package `javax.swing`	newer classes dealing with GUIs

These package names act as paths on your hard drive. From the above package names, we can see that there are two main directories, `java` and `javax`, and within directory `java`, there are directories `io`, `net`, and `awt`. Below, we show this directory structure, giving a few classes that are in directory `awt` and `swing`.

```
java
   io
   net
   awt
      Frame
      Button
      Textfield
javax
   swing
      JFrame
      JButton
```

You can form your own packages of classes that you write, but we don't show you how to do that here.

The import statement

In the interactions pane of DrJava, let's type the assignment statement

j= new JFrame();

and hit the enter key. You see that there is an error: class JFrame is undefined.

Now let's type the import statement

import javax.swing.*;

This statement gives your program access to all classes in package javax.swing, without needing to give the complete path. The asterisk indicates "all classes" in the package.

So, if we type the assignment statement again, there is no error, and we can also show the new JFrame — there it is.

If you want to "import", or make accessible, only one class in a package, then use its name instead of the asterisk. For example, after importing java.awt.Frame, we can access it but no other class in package java.awt.

You have seen the use of the import statement in the interactions pane. When we begin writing classes, we'll show you where to put it.

Packages

The class definition

We write our first Java class, in the upper right pane of DrJava. This class definition will describe the components that are placed in each manilla folder, or object, of the class. We start by putting a comment that indicates what each instance of the class has in it.

/** An instance is like a JFrame but has two more methods */

We then type **`public`**, which means that this class can be referenced from anywhere else; **`class`**, which means that we are writing a new class definition; a name for the class, we choose `Demo`; and an opening and closing brace. The opening brace goes on the same line with the name of the class.

The words **`public`** and **`class`** appear in blue, because they are keywords and cannot be used as identifiers —as names of variables, methods, or classes.

Instances of class Demo have essentially nothing in them, because we haven't declared any components.

We save this class definition, by clicking button `Save`. A navigation window opens, and we navigate to where we want to place the program, create a new folder, and save the program with the same name as the class. *Every program, consisting of a bunch of classes, should be placed in its own folder*!

The name of the file in which this class definition was saved, `Demo.java`, now appears in the left pane. And we also see file `Demo.java` in the righthand pane.

Here is a rule to follow:

/* **Rule for file names**: Each public class definition should be

placed in its own file; the name of the file should be name

of the class, and the extension should be `java`. */

Let's look in the folder to see whether the file is there. Yes, there it is.

Compiling the class definition

Now, we compile the class definition, by clicking button `Compile`. This translates the class definition into a machine language, so that it can be executed. Look in the folder that contains `Demo.java`. There is now a file `Demo.class` —this file contains the machine language version of class `Demo`.

Creating objects of the class

We can now create an instance of the class, by typing:

```
j= new Demo();
```

in the interactions pane. Notice the use of a new-expression to create an object of class `Demo`. But there are no methods in it, as far as we know, because we did not define any in the class definition. For example, method `show` does not exist, which we can verify by typing `j.show()` into the interactions pane.

However, we can ask for the value of expression `j` —its value is the name on the tab of the manila folder! It consists of the name of the class, an @ sign, and then some other characters.

Extending class JFrame

Now, we would like instances of class `Demo` to have all the fields and methods that an instance of class `JFrame` has. We do this in two steps. First, we put an *import statement* before the class definition:

import javax.swing.*;

This import statement indicates that all the classes in package `javax.swing` may be used.

Next, after the class name, we put an *extends-clause*, indicating that class `JFrame` is to be extended. Note that **`extends`** is a keyword. We compile the program again and then execute the assignment of a new instance of class `Demo` to `j`. Now, when we execute `j.show();`, the `JFrame` window appears on the screen! This is evidence that the use of the extends-clause has caused all the components of `JFrame` to be *inherited* —they

actually appear in each folder of class `Demo`. We can set the title of the window using procedure `setTitle`, and we can get the width of the window —we can use all the methods that appear in every `JFrame` object.

In the next lecture, we show how to draw a manilla folder of class `Demo`, and we show how to define a function and a procedure in class `Demo`.

The subclass definition

The manila folder for subclass Demo

Here is a manilla folder —an object or instance— of class `JFrame`. The tab contains the name of the folder, the name of the class appear in the upper right, and we show only some of the methods and none of the fields in the folder.

The definition of class `Demo` *extends* `JFrame`. We say that `Demo` is a *subclass* of `JFrame` and `JFrame` is the *superclass* of `Demo`. Remember these terms.

We draw a folder of class `Demo` with two partitions. The top partition is labeled `JFrame`; it contains all the components —fields and methods— that a `JFrame` has. We say that `Demo` *inherits* all these components. The bottom partition is for components that are defined in `Demo` itself. There are none yet.

The `Demo` partition appears below the `JFrame` partition because `Demo` is a subclass of `JFrame`.

Defining a function

Let's define a function in class `Demo` that yields the area of the window in pixels —its height times its width. This is your first Java function definition. We start off with its specification, as a comment. It says that a call of this function equals the area of the window.

Now we put the header of the function. Keyword **`public`** means that it can be called from any part of the program; the function will return an integer, an **`int`**; and we give the function the name `area`. Parentheses are needed after the method name.

Then comes the *body* of the method, which is enclosed in braces. Note that we put the opening brace on the same line as the header of the method.

In the body of the method, we put a *return* statement, giving keyword **return** followed by the expression whose value is to be returned followed by a semicolon. And that completes your first function definition:

```
/** = area of the window of this JFrame */
   public int area() {
      return getWidth() * getHeight();
   }
```

Let's compile the class again. Since the definition is now in the class, we write the new function in the lower partition of each instance of `Demo`. To impress upon you that the whole method appears in the object, and to make a point in a moment, we have written the complete method body —we'll erase it later.

The method body calls function `getWidth`. How does it know where this method is? Since `getWidth` is not defined in the method body itself, we simply look outward, and upward, and find it in the `JFrame` partition, so that is the `getWidth` function that is called. This idea of looking from the inside out to enclosing constructs will be explained in more detail later on.

Watching a call on the new function

Now let's create a new instance of class `Demo`, store its name in variable `j`, show the window, drag to make it bigger, and call function `area` using `j.area()` —look, it worked! You have seen your first call of a function that you wrote.

Defining a procedure

A function returns a value; a procedure does not. Instead. The procedure does something. Let's write a procedure that sets the title of the window to the area of the window, so that the bottom partition contains two methods.We start by writing a specification of the procedure:

```
/** Set title to "Area is " + <area of window> */
```

Now, we write the header of the procedure. We start with **public**, so the procedure is available everywhere. Since this is a procedure, in place of the "return type" that is used in a function, we write **void**. We choose `setTitleToArea` for the name of the procedure. And we need the parentheses and then the braces for the procedure body.

The subclass definition

The body itself will be a single statement that calls procedure `setTitle`, with an argument that gives the desired string —notice how we call function area.

```
/** Set title to "Area is " + <area of window> */
public void setTitleToArea() {
    setTitle("Area is " + area());
}
```

We compile the program, store a new instance of `Demo` in `j`, and show it. Let's drag to make it bigger and then call procedure `setTitleToArea`. And let's do it again to see that the area actually changes.

Summary

You have seen a function definition and a procedure definition. The function definition has the *return type* —the type of value the function yields— while the procedure has the word **`void`** in its place.

The function body must have as its last executed statement a return-statement, which gives the expression whose value is to be returned. The procedure body need not have a return-statement. Its body consists of a sequence of statements to be executed.

The name of a function is generally a noun (or noun phrase) that tells you what is computed (like `area`, or `sqrt`). It may have the word `get` in front of it. The name of a procedure is usually a command to do something. Both functions and procedures generally start with small letters, and if they consist of several words, the first letters of all but the first word are capitalized.

We will return to a study of functions and procedures later on.

You have also seen what an instance of a subclass looks like —it has a partition for the superclass and, underneath it, a partition for the subclass. The bodies of methods defined in the subclass can reference inherited methods and fields directly.

Field declarations and getter/setter methods

Consider writing a class `Chapter`. Each instance of `Chapter` will contain information about a chapter of a book, like the chapter number, so we need to declare a variable in the class. The basic form of a declaration of this variable, as you know, is **`int`** `number`, but we put a semicolon after it:

```
int number; // The chapter number.
```

This is what an object of class `Chapter` now looks like —yes, it is initialized to the default value 0.

A variable declared in a class definition is called a *field* or *instance variable*, because it is appears in every instance of the class. And, you can see that we can create an instance of the class, assign a value to the field, and tobtain the value:

```
x= new Chapter();
x.number= 3;
x.number
```

Access modifiers for fields

Generally, we give an *access modifier* in the declaration. Access modifier **`public`** allows access to the field from any place in which the instance is accessible. Instead, we use access modifier **`private`**, which signifies that the field is accessible only from code within class `Chapter`.

Here, we'll show you. Compile the program, create an instance of it, as before, and then try to assign field `number` a value —Java complains; it is not allowed. Even an attempt to obtain the value of the field leads to an error message: IllegalAccessException: cannot access a member of class `Chapter` with modifier **`private`**.

In software engineering, we use the general rule that fields of a class are **`private`**. There will be exceptions to this rule, but from now on, make fields of classes **`private`**.

Let's also add a declaration of a field to contain the title of the chapter —notice how we put a comment that gives the meaning of the variable. The default value in a field of a class-type is **`null`**.

```
private String title; // The title of the chapter.
```

Getter functions

We have a problem: we can't look at or change the fields, so what good are they? To solve this problem, we write two kinds of methods.

First, we introduce *getter* methods, or functions, to *get* the value of the fields. Note how we start off with a specification, which says what value the function returns. And we use the standard Java convention that the name of the function is the name of the field preceded by the word `get`. We compile, to make sure there are no syntax errors. We compile *very* often. Let's put in a getter function for field `title`, too —we'll do it by copying the first, pasting, and editing. We compile.

Finally, we use the interactions pane to show that we can now access the the fields of an instance using he getter functions.

Setter procedures

To be able to store values in the fields, we define so-called *setter* methods, or procedures. We put in a specification for a procedure to store a value in field `number` —note that it is a command to do something. We put in the header of the procedure: it's **`public`**; it's a procedure, so we use keyword **`void`**; the name is the word `set` followed by the name of the field; and we put a *parameter declaration* within parentheses to indicate the variable, `n`, that will contain the value to be stored in field `number`; and we end with the braces for the procedure body. Compile to make sure we haven't made a syntactic error.

Finally, we put an assignment in the body to assign parameter `n` to field `number`. And compile. And now we show in the interactions pane that we can use the setter procedure to assign a value to the field and then use a gett function to retrieve the value.

To complete our work, we put in a setter procedure for field `title`.

Field declarations and getter/setter methods

About parameters and arguments

This is the second method you see that has a parameter. Later, we will go into more detail about how an argument value is associated with the parameter when the method is called. For now, we slip by this issue, assuming that you have seen function, procedures, and calls on them in another programming language.

Discussion

We made a field **`private`** so that it couldn't be accessed. We then added a getter method and setter method to provide that access! Later, we will see the software-engineering reasons for allowing access in this fashion. You don't *have* to provide either the getter method or the setter method. If you don't want to allow access to a field, don't provide a getter method. If you don't want to allow someone to change a field, don't provide a setter method.

So, for now, accept that fields should be made **`private`** and write getter/setter methods if you want.

Constructors

Here's class `Chapter`, with three fields, the chapter number, chapter title, and the previous chapter. Here is an instance of that class, created using an assignment `c= new Chapter();` .

The only way to initialize the fields of the new instance is through calls on the setter methods, for example:

```
c.setNumber(1);  c.setTitle("Intro");  c.setPrev(null);
```

If we had more fields —and many classes will have 5 or 10 or even more— it would become more laborious and awkward to initialize the fields. We need a simpler way to initialize fields of an new object.

Java lets us indicate the initialization in the new-expression itself. Instead of having to create the instance and then initialize the fields, we will be able to do it like this —the 1 is the value for field `number` and the string is the value for field `title`.

```
c= new Chapter(1, "Intro");
```

We won't have to initialize field `prev` because that is initialized to a default value of **`null`**, anyway.

But this doesn't work yet!. Before it will work, we have to write a new kind of method, called a *constructor*. Here's how we do it. First, write a specification:

```
/** Constructor: an instance with chapter number n,
                 title t, and previous chapter null */
```

Then, we write the constructor: it will be **`public`**. Since it is a constructor, we do *not* put a type or **`void`**. Since it is a constructor, its name is the name of the class. And we put parameter declarations for the parameters that will contain the chapter number and the title. Let's compile to make sure that the syntax is correct.

Now, write the constructor body: assign parameter `n` to field `number` and parameter `t` to `title`. We don't have to assign anything to field `prev` because its initial value is **`null`**, anyway. We compile.

This constructor now appears in each instance of the class, just like the other methods.

Let's see whether it works. Create an instance, giving argument values for fields `number` and `title`, and store the instance in variable `c`. Then, access all three fields. It works!

```
c= new Chapter(1, "Intro");
c.getNumber()
c.getTitle()
c.getPrev()
```

About constructors

Now let's give the rules for constructors. First, the constructor is a new kind of method. It has a purpose, which you should memorize:

> **Purpose of a constructor**: to initialize some or all of the fields of the newly created object during evaluation of a new-expression.

Next, the constructor definition includes:

1. An access modifier —usually **`public`**, so that the constructor can be called from anywhere.
2. The name of the constructor , which is the same as the name of the class! Notice that neither a type nor keyword **`void`** appears in a constructor definition.
3. One parameter declaration for each field for which a value is to be given in the new-expression.
4. A method body, which assigns each parameter to the appropriate field. Here, **`int`** parameter `n` is stored in field `number` and `String` parameter `t` is stored in field `title`.

The constructor, then, is the third kind of method. It looks different from a function or procedure because it does not have a return type or keyword void and because its name is the name of the class. Now, let's see how to call it.

Constructors

A new look at the new-expression

We can now give a complete explanation of the new-expression, in terms of class `Chapter` and this call:

```
new Chapter(1, "Intro")
```

You can read the general discussion in Gries/Gries, Section 3.2.2, p. 116.

We describe the three steps in evaluating a new-expression —this is something you should memorize and be able to perform on your own.

Step 1. *Create a new instance of class `Chapter`, with default values for the fields*.

The default values for fields are 0 for integral types, 0.0 for the floating point types, **`false`** for **`boolean`**, and **`null`** for all class-types.

Here is the new object, with all methods except for the constructor *not* shown.

The beginning of the new-expression helps you remember step 1, for it says "new Chapter"!

Step 2. *Execute the constructor call of the new-expression* —in this case, `Chapter(1, "Intro")`.

We know that this call will assign 1 to `n` and `"Intro"` to `title` and then execute the two assignment statements, making `number` 1 and `title` `"Intro"`.

Also in this step, the form of the new-expression tells you what to do. There is a constructor call; execute it.

Step 3. *Yield as value of the new-expression the name (on the tab) of the new object*. In this case, the value of the new-expression is `a0`.

Conclusion

And that is how a new-expression is evaluated. You will fully understand what is going on only when you have memorized the three steps and can carry them out yourself, as we have done here. The computer doesn't have to evaluate a new-expression. *YOU* can do it too.

This is evaluation is so important for your understanding that we ask you to take a quiz on it.

Function toString

Function toString for class Chapter

In the DrJava right upper pane, you see class `Chapter`, with three fields: the chapter number, chapter title, and the previous chapter. Along with two constructors and getter/setter methods. We often want to be able to get a description of an object of such a class —we want to see the values of the fields in a nice format, to make clear which value is which.

The Java convention for obtaining a description of an object of a class is to write a function `toString` in the class. Here is a standard specification for it, although one might go into more detail in some cases. It's **public**, it returns a `String`, its name is `toString`, and it has no parameters. We insert a return-statement, so that the class can be compiled, and compile.

```
/** =  a representation of this chapter —for the previous
       chapter, the chapter title is given. */
public String toString () {
    return "";
}
```

Now let's write the real return statement. Let's start off with the word *Chapter*; catenate to it the chapter number, a period, and a blank; and catenate to that the title and another period and blank. For the previous chapter, we have to be careful. We write a conditional expression; if the previous title is null, we append "No previous chapter", and if it is not null, we append the title of the previous chapter, as required by the spec.

```
return "Chapter " + number + ". " + title + ". " +
       (prev == null ? "No previous chapter"
                     : "Previous chapter title: " + prev.title);
```

Now let's check it out. We create two objects of class `Chapter` —let's make sure one of them has a non-null previous field—and call their `toString` functions. See?

```
Chapter c1= new Chapter(1, "Intro");
Chapter c2= new Chapter(2, "Truth", c1);
```

And that's all there is to writing a function `toString`. Generally, the result contains all the fields, but there are times that that will be too much, and a less complete `toString` function will be satisfactory.

Funtion toString for class Point

We write a function `toString` for class `Point`, an instance of which contains a point `(x, y)` in the plane —`x` is the horizontal coordinate and `y` the vertical coordinate. We have the two fields, `x` and `y`, and a constructor, which allows both fields to be initialized in a new-expression. We haven't written other methods because they are not needed for this demoinatration.

Here comes function `toString`. First, we put in a specification for it; note how the spec tells us what the represention will be. The function is public; it returns a `String`; its name is `toString`, and it has no parameters. We put in a return statement in so that we can compile.

```
/** = a representation "(x, y)" of this point */
public String toString() {
    return "";
}
```

We write the real return statement. We start with the opening parenthesis; catenate field to it; catenate to that a , and blank; then field y, and finally the closing parenthesis.

```
return "(" + x + ", " + y + ")";
```

That wasn't hard! Let's construct a few objects of class Point and see what their toString function produces.

```
(new Point(4, 5)).toString()
(new Point(0, -3.2)).toString()
```

Function toString

Automatic use of function toString

Suppose an object appears as an operand of a catenation, like this:

```
p= new Point(3, 5);
"Here is the point: " + p
```

In this situation, function `p.toString` will be used in place of `p`, as you can see when Drjava evaluates this expression. This allows us to abbreviate and keep things simple.

There is another reason for writing `p` instead of `p.toString` in these cases. If `p` contains **`null`**, evaluation of `p.toString` in

```
"Here is the point: " + p.toString()
```

causes an error, but evaluation of `p` in

```
"Here is the point: " + p
```

does not —the value **`null`** is simply changed into a `String`.

Another place where toString is not necessary is in a println statement, as you can see from execution of this statement:

```
System.out.println("A point: " + new Point(1,2));
```

Conclusion

The Java toString convention is built into the language, in that in several places, the use of an object will automatically result in its function `toString` being called. Make heavy use of the Java toString convention. Further, whenever you write a new class, immediately after writing the constructors, write function `toString`! Doing so will save you testing and debugging time, because you will be able to look at the descriptions of the various objects that your program creates.

Functions toString and equals in class Object

Function toString in class Object

Function `toString` in class `Object` is defined to return the name of the manilla folder, or object in which it appears. For example, evaluate this expression in DrJava's interactions pane:

```
new Object()
```

and it will evaluate to something like `java.lang.Object@d70b42` —that's the name on the tab of the manilla folder or object, like the `a0` or `a1` that we have been writing from time to time. The name consists of the name of the class (including its package information), the `@` sign, and then 6 hexadecimal digits, whose meaning you don't have to know about at this point.

Object's function toString is inherited in every class but is usually overridden

Since `Object` is the root of the class hierarchy, its function `toString` is inherited by every class, so the function appears in *every* object.

However, if you define function `toString` in a class, your definition overrides the inherited one. So, the `toString` function that we wrote in class `Point` in a previous web lecture overrode the `toString` function that was inherited from class `Object`. So, a call `p.toString()` calls the overriding function.

Later, in module 2, you will see that you may still be able to call the inherited function.

Function equals in class Object

Now let's turn to a discussion of function `equals`, which is also defined in class `Object` —like this:

```
/** = "this object and object ob have the same name on their tabs
       (i.e. since object names are unique, they name the same object)". */
public boolean equals(Object ob) {
    return this == ob;
}
```

We have shown the whole method in the object, to make sure that you understand that it really is there. You may not have seen a use of keyword **`this`** before. It refers to the name of the object in which is occurs, in this case, so it refers to `a0` itself. Therefore, a call on this method tests whether ob contains the name `a0`.

You can see how function `equals` works by evaluating this sequence in DrJava's interaction pane.

```
ob1= new Object();
ob1.equals(ob1)            // This is true.
ob1.equals(new Object())   // This is false, because ob1 and the new object are different.
```

Function `equals` is often overridden in newly defined classes, with the convention is that `equals` in a newly defined class should test for equality of all fields of the class. We will return to a discussion of overriding function `equals` in module 2. But for now, we concentrate on testing equality of strings.

Testing for equality of strings

It is a big *mistake* to use == to test for equality of `String`s, because a `String` value is an object, and not a value of a primitive type. For example the expression

```
"1" == "" + "1"
```

evaluates to **`false`**, even though both sides have the `String` `"1"` as their value, because == tests whether the names of objects, and not their contents, are equal. You can see this with the two objects whose names are in s0 and s1.

The moral of the story is: *do not use == to test for equality of strings*!

But class String does override function `equals`, with a function that tests for the equality of actual strings and not of object names. The following function calls all evaluate to true:

Functions toString and equals in class Object

```
"1".equals("1")
"1".equals("" + "1")
"1".equals("" + (2-1))
```

Using == with strings is a mistake that all Java programmers have made at least once. If you make the mistake, you will find that it's a tough bug to find —unless you are aware that it might happen and can look out for it. So be aware of the problem with using == for equality of strings.

Specifications of methods

You know what the first function `mini` does, because its specification, in the comment preceding it, tells you. You don't need the function body —which we show you now.

You have no idea what the second function does, because its specification has not been given. To understand what it does, you would have to look at its body: `Integer.signnum`.

A specification of a method defines what a call on a method does. The specification should be precise, complete, and as clear as possible. So that you don't have to look at the method body.

In fact, *every* method should have a specification, and you should *never* have to look at the method body when writing a call on a method. The method body indicates *how* a task is carried out. When writing a call on a method, we don't want to know that, we want to know *what* the task is. And the spec should tell us.

Writing procedure specifications

We write method specs as *Javadoc comments* immediately preceding the method. Javadoc comments begin with `/**` and end with `*/`. The next lecture will explain why we use Javadoc comments.

Let's begin discussing specs for procedures. There are two points to make here.

```
/** Set the title to t */
public void setTitle(String title){...}
```

1. The spec mentions every parameter by name, and in a way that explains what that parameter is for. If a spec doesn't mention a parameter, the reader may not understand what the parameter is for, and the spec is incomplete.

2. This spec is a *command* to do something. We don't say, "this procedures sets the title to t", or something similar. We say, "Set the title to t." This is because we use it as follows. Suppose we have a call on procedure `setTitle`: `setTitle("I want peace");` . The way to figure out what it means is to replace it by the specification, but substituting the argument for the parameter:

```
Set the title to "I want peace".
```

Since the procedure call is a command to do something, so should be the specification.

Writing function specifications

Since a function returns a value, the spec should be a description of that value. In fact, it should say what a call on the function equals. For example, that's what the spec of function `areAscending` does. It says that a call `areAscending(a, b, c)` is equal to the value of the true-false sentence "`a`, `b`, and `c` are in ascending order."

```
/** = "a, b, and c are in ascending order".*/
public static boolean areAscending(int a, int b, int c){...}
```

Since functions evaluate to a value, we often write their specifications starting with the equals sign. You may see a spec like "Return true if a, b, and c are in ascending order." This has a sort of operational feel, which we would rather not have. In mathematics, if someone asks, "what is sin(0)," one doesn't say, "return the value of the sin of 0,", one says simply," that's (equal to) the sin of 0. Same thing with any function call.

Writing constructor specifications

Here's how we write a specification for a constructor. The word "Constructor" announces that is is a constructor. And it explains what the properties of the new instance —what each field contains. Notice that the spec says what the previous chapter will be, even though there is no parameter for it. Note also that all parameters are explained:

```
/** Constructor: a new instance with chapter number n, chapter title t, and previous chapter null. */
public Chapter(int n, String t) {...}
```

Good specifications

Specifications of methods

Get in the habit of writing a method specification *before* you write the method body. For two reasons. First, you won't forget to write the spec, Second, that spec will help *you* the programmer, as you write the method body. You will be able to refer to the specification as you write the body. You don't have to remember everything about the spec.

Make sure that your spec is accurate and complete. It must be consistent with the method body. A spec that is wrong in any way is probably worse than no spec at all, because a programmer who relies on the specification in writing a call on the method is bound to be introducing an error into their program. [Adam: do NOT say "his or her". Say "their". After reading this, delete this comment.]

Finally, include explicitly any *preconditions* on a call —that is, constraints that the caller must satisfy. For example, the specification of a function that computes the square root of r should include the statement:

Precondition: r >= 0.

Such a precondition need not be checked the function; it is up to the caller to call the function only when the precondition is true.

Changing a spec

If you realize you need to change what a method does, first change the specification, and *then* change the method body. Be disciplined in doing this, and you will make fewer errors.

Let's show how this works. Here's our first step: changing the specification.

And here's our earlier function `areAscending`, which may not treat equal parameters the way we want. For example, if a = b < c, true, and not false, should be returned. So, we change the spec to:

`/** = "a, b, c` are in non-descending order `(a <= b <= c)" */`

We have revised the spec. The next step is to revise the function body to be consistent with the spec, and here it is.

Javadoc specifications

A programmer who wants to use your program should be able to look at the specification of the program, its documentation, without having to look at the code itself. Java has a facility for extracting documentation from your program —provided that you have written your documentation appropriately. This requires placing *javadoc comments* before each class, field declaration, and method declaration. These comments have the form:

```
/** ... */
```

To show you how this works, look at class `Elephant` in DrJava. And here is the folder in which file Elephant.java appears. Notice that we have put an empty directory doc in it. We suggest you do this also.

Now, return to the DrJava application, and click item **Javadoc** on the right of the tool bar at the top of the DrJava window. After waiting a while, a browser window will open, which contains the javadac spec in it, in the same format as the Java API specifications. Look at that! Take a look at directory `doc`; you will see lots of files, all of which were produced by hitting the javadoc button.

Now let's look at the specification in detail.

1. Note that it has a page for class `Elephant` and that it extends `Object`. Look at the comment for the class; it is precisely the javadoc comment that appeared just before the class definition.

2. *The Field Summary part* contains a list of public fields, with the javadoc comments that were extracted from them. Note that only the first sentence is given —that is, up to the first period. If you click on the field name, you get the "field detail", and the whole extracted comment will appear there.

Let's go back to the summaries.

3. *The Constructor summaries* give information about all the constructors, again with the javadoc comments that you put on them.

4. *Here are method summaries*. Note that there is no comment for procedure `setName`. Why not? Let's go back to the program and look at the comment that is there. Aah! It is not a javadoc comment because there is only one asterisk. Javadoc comments require *two* asterisks, as in the spec of function `getName`.

As in the field summary, in the method summary, we can click on the name of a method to get to the method detail.

5. Finally, you can see methods that are inherited from superclasses, in this case, class `Object`. We can click on `equals` to get to its method summary and then on its name to get to the method detail. Hey, notice how it says that `equals` should be reflexive, symmetric, and transitive. Let's go back.

You may see a navigation window

When you click the javadoc button, a navigation window may appear, which asks you to select a directory into which the Javadoc files will be placed. We always select directory `doc` within the folder for the project. After you select the folder, DrJava creates the Javadoc files —be patient; it can take a few moments. Finally, a new browser window will appear with the javadoc spec in it.

Your duty as a Java programmer

It's your duty as a Java programmer to write javadoc specs for *all* public components. It is also your duty to click the javadoc button and to read *all* the extracted specs carefully to make sure they make sense, are precise, are thorough. If the purpose of a method call cannot be understood from the method spec, then there is something wrong with it. Here are points to consider.

Since javadoc documentation is meant mainly for users of classes, javadoc generally extracts specifications only for public classes, methods, and fields —those that the user can use. However, in the preferences, you can set a switch that forces javadoc to extract javadoc comments from private entities as well.

1. *The javadoc spec for a class* should explain briefly what an instance of the class is. For example:

```
/** An instance of the class is a shape, like a polygon. */
```

2. *The javadoc spec for a variable* should state the meaning of the variable and any constraints on it.

3. *The javadoc spec for a method* should be a precise and thorough spec, as we have explained elsewhere. If one cannot tell how to write a call on the method from that spec, then the spec is no good.

Testing

A *bug* is an error in a program. Testing is the process of analyzing and executing a program to determine whether it has bugs. Our main tool in testing is the *test case*: a set of inputs together with the expected output.

Debugging is a separate process. Having detected that the program has a bug, we need to find out what caused it and remove it. This process is called *debugging*.

Principles for creating test cases

In this lecture, we concentrate on test cases —how to develop them, when to develop them, and why. Writing a bunch of test cases, as we ask you to do in assignments, may feel like wasting time. But if you do this task properly, you will find that your productivity will increase.

Developing good test cases is important. Some companies have quality assurance (QA) teams that do nothing but test software —other people, not the original programmers, do testing once the programmers believe their product is finished.

When both writing and testing a method, understand exactly what the method is supposed to do. This means that *the method specification should be written before the method body.* Some people advocate also writing a set of test cases before writing the method body, for this helps us understand what the method is supposed to do. This can be useful when writing a method from a spec that someone else gave us.

Test early and often. The sooner you test a method, the sooner you will find and fix bugs. In fact, as soon as you have written a method specification, you should write and example of a call to that method, so that you will be thinking about how the method might be used even before you write the method body.

Think about the whole set of inputs. It is not always feasible to test every single value of input because set of input can be infinitely large such as set of integers, so we need some kind of strategy to make sure although our test cases do not include the entire of inputs; they are enough to guarantee that our program will work properly. We will give you an example of set of order pair (x, y) of integer here to illustrate this principle.

First, we will visualize the set of order pair (x, y) as a square. There are about 2^{64} points in this square. Testing all of them will unnecessarily waste of time, so we need to find a set of points in this square that is in interesting regions. In this case, we may want to first test a few points in all four quadrants such as (15, 17), (-10, 20), (-100, -200), or (134, -27). Then, we may want to try some extreme values or *corner cases* such as -2^{31} (minimum integer), or $2^{-31} - 1$ (maximum integer), or 0. There are other interesting values we may forget as well such as 1, or -1. We can also apply this idea to different sets of inputs. For example, if our set of inputs is a class, the extreme value will be **`null`**.

Develop test cases that provide test coverage. A group of test cases provides test coverage if exercising them will cause every single part of the program to be executed at least once.

Test only one thing at a time. If you test several things at once and the test fails, you will have a harder time figuring out why. Also, it is easier to lose track of what you are testing and harder to make sure that you have covered all the cases.

Test each method thoroughly as it is completed. Once a method is tested and its correctness is assured (as much as possible), you will be quite sure that any error that arises is not in that method. It lies elsewhere. This reduces, to a large extent, the area of the program to be investigated when developing test cases, testing, and debugging.

Verify the documentation. As you test, ask yourself questions that you would not normally think of while writing the code. Make sure the documentation answers those questions.

Approaches to creating test cases

Exhaustive testing. Testing a program on all possible inputs, sounds good but is generally infeasible for most programs because of the number of possible test cases.

Blackbox testing, or functional testing. In making up test cases, one looks only at the specification of the program (not the program itself) in deciding what test cases to try. The program is a black box, and you cannot see inside it.

Glassbox testing, or structural testing. One looks at the program itself when developing test cases and uses the structure of the program to recognize possible trouble spots and develop test cases to exercise them.

Testing a class

Testing a class is harder than testing a method because it generally contains many methods. This is what we should do when we test a class.

Declare at least one and perhaps more variables with the class as their type. Then, create instances of the class and assign them to the variables.

Check that the constructor of the class works properly. This means checking whether the fields of each new instance were properly initialized. Function `toString` can be useful in testing. If `toString` has been written, it is easy to print the contents of a class instance using it:

```
System.out.println(x.toString());
```

This test will also help test method `toString`.

Test all the other methods. This testing can follow the principles described earlier.

JUnit Testing

DrJava has built-in mechanism that makes it easy to create a class to maintain a suite of test cases and to run *all* test cases whenever appropriate just by clicking a button. The mechanism, which is used in many interactive development environments, is called JUnit.

We now show how to construct test cases using a JUnit tester.

Class Panda

In DrJava, we have a class `Panda`, each instance of which maintains information about a panda. To keep things simple here, the class maintains only the panda's name and its father, if the father is known. In a real situation, much more information would be maintained. There is a constructor, two getter methods, and two setter methods. They must all be tested, and we do this using a JUnit testing class that saves all test cases.

Creating a JUnit testing class

To create a JUnit class, select menu **File** item **new JUnit test case…**. At the prompt, type a class name for this tester. We recommend the name of the class being tested followed by the word "Tester". Click button *OK*.

The new class has been created. Save it in the same folder as class `Panda`.

The import statement indicates that a class `TestCase` appears in package `junit.framework`, and class `PandaTester` extends `TestCase`. Several methods are inherited from class `TestCase`. We'll show the use of one of them later.

Class `PandaTester` comes with one procedure, `TestX`. As its comment says, we can replace the `X` by our own name, and we replace it by `ConstructorGetters`.

We can have as many test procedures as we wish. We add a second one for testing the setter methods.

Writing a test method

Now let's write the body of the procedure to test the constructor and getter methods. First, we need to create some Panda objects; let's use these two assignments:

```
Panda p1= new Panda("Shuaung", null);
Panda p2= new Panda("Lin", p1);
```

Note that we have created one without a father and one with a father in order to test these two different cases.

To test whether object `p1` has been correctly created, we need to check that each field of `p1` has the right value. Inherited procedure `assertEquals` can be used for this purpose. This procedure tests whether its two parameters are equal. The first parameter is the expected value, and the second is a value that is produced by the method being tested.

Here, we write two calls on `assertEquals`, testing whether the getter methods in object `p1` produce the right value:

```
assertEquals("Shuang", p1.getName());
assertEquals(null, p1.getFather());
```

In the same way, we insert statements to test the getter methods of p2:

```
assertEquals("Lin", p2.getName());
assertEquals(p1, p2.getFather());
```

Running the test cases

Running the test cases is easy. First, compile the program and wait for button *Test* in the right half of the button bar of DrJava to become black. Then click it. The names of the procedures that begin with "test" appear in green in the bottom pane, and there is a green bar labeled "Test Progress". This means that executing calls on the test procedures detected no errors.

To show what happens when there is an error, let's change the expected value of the first call of `assertEquals`, compile again, and click button *Test*. Now, the name of the test procedure appears in red, meaning that an error was detected, and the offending call on `assertEquals` is highlighted.

We fix the error, compile, and test again. Now the tests work.

Testing the setter methods

We test the setter methods in the same way, putting the test cases —implemented in the `assertEquals` calls— in procedure `testSetters`. We write code to create one panda. And we write code to call each setter method and check whether it sets the field properly. Note that we make panda `p1` be its own father. It's ok, it's just for test purposes.

```
Panda p1= new Panda("Shuang", null);

p1.setName("Harry");
assertEquals("Harry", p1.getName());

p1.setFather(p1);
assertEquals(p1, p1.getFather());
```

Now compile and run the suite of test cases by clicking button *Test*. Voila! Everything is alright.

A few comments

A few comments are in order.

1. First, there is no need to put a specification on these test procedures, as long as the name of the procedure gives some indication of what is being tested.
2. Second, every method whose names begins with `test` will be called when button *Test* is clicked.
3. Third, note that there are many `assertEquals` procedures, one for each primitive type and one for each class type as well. You don't have to think about which one you are calling; just write the calls.
4. Fourth, in a call of `assertEquals`, the expected value is the first argument and the computed value, which should equal the expected value, is the second argument.
5. Fifth, how many different test procedures you write is up to you. Writing one test procedure for each method to be tested would be too much work. Don't do it! On the other hand, putting all test cases in one test procedure can make that test method too long and unwieldy. Don't do it!

   Instead, try to find a proper balance, by placing methods to be tested into logical groups —for example, getter methods, setter methods— and writing a test procedure for each group, so that no test procedure is too long and that they are all easily maintained.

Debugging

Introduction

Testing is the process of running the program against test cases to try to uncover errors. *Debugging* is the process of looking for the cause of an uncovered error and then fixing it —once the cause has been found. Often the cause is far removed from the place at which the error manifests itself.

Some people hate debugging, seeing it as a long, arduous, and frustrating task, like looking for a needle in a haystack. Others enjoy it, seeing it as a problem solving activity that requires the skill of a detective, who looks for evidence and sifts through it, trying to find out who committed a crime.

There are two basic methods for tracking down a bug:

1. *Use the debugging feature of your IDE*. In most IDEs, you can turn on the "debugger", which allows you to step through execution of the program one statement at a time, watching how the values of the variables change. You can also set "breakpoints", and then let execution run its course, and it will stop whenever a breakpoint is reached and let you investigate values of variables.
2. *Insert print statements at judiciously chosen places of the program*. Then execute the program and analyze the output.

In this course, we concentrate on method (2), inserting print statements. The next item asks you to watch a few lectures on the CD *ProgramLive*. These lectures will step through the process of debugging a program. Some part of the program calls a function `anglicize`, which translates an integer into its English equivalent —e.g. 123 is translated into "one hundred twenty three". Function `anglicize` doesn't seem to be working properly, and debugging takes place to find, "out why.

Debugging maxims

ProgramLive, as well as the text Gries/Gries, present four maxims that help in tracking down bugs. Because they are important, we state them once more here:

- *Debugging maxim 1*. Place print statement at the beginning and end of the method that appears to have an error in it —to check whether its precondition is met and to check whether it returns the correct value.
- *Debugging maxim 2*. Glean as much information as possible from each exercise of test cases.
- *Debugging maxim 3*. Insert print statements at judiciously chosen places, in order to check values of variables.
- *Debugging maxim 4*. Deal carefully with interfaces, for they are the source of many errors.

Finding API specs on the web

The Java API specifications describe what each class that comes with Java is for and how it is to be used. These specifications are on the world wide web, so you have to be connected to the internet to use them. You can download the specs, but they take a lot of space. If you can usually work with a connection to the internet, you are better off doing that rather than downloading.

Where to find the Java API specifications

At the time of this lecture, Java Standard Edition 6 has been released. Its specifications are at this URL:

`http://java.sun.com/javase/6/docs/api/`

Bookmark it! If you forget it and don't have access to the course website, google something like "Java API".

Components of the Java specification window

Visit the API specs in some browser. The small pane on the upper left contains a list of the Java packages, and the pane on the lower left contains a list of all classes in those packages (as well as items called interfaces). Both panes are scrollable, so you can easily find what you are looking for.

The main pane on the right contains a horizontal button bar, whose items are self-explanatory. The lower part of the main pane contains a list of packages in the Java API. The word *deprecated* means 'lessened in value'. Java uses the term for classes and methods that have been superceded by better ones. You can still use the deprecated ones.

How to access the spec of a particular class

Let's access class `Math` in package `java.lang`. First, use the scroll bar in the upper left pane to make `java.lang` visible, and click on it. The lower pane changes to show only the interfaces and classes in that package, making it easier to find what we want. Use the scroll bar in the lower pane to make `Math` visible and click on it. The description of class `Math` appears in the main pane.

Scroll down slowly, and you will see a "summary" of the fields and methods in class Math. These summaries tell you a little about the fields and methods of the class. For example, here, the first entry summarizes function `abs`, and we see that it has attribute **`static`**, that it produces a **`double`** value, and that it has a **`double`** parameter (named `a`). Also, it returns the absolute value of a **`double`** value.

Click on the underlined word `abs`, which is a link to another part of the same web page. The main pane changes to display a more extensive description of the method. This description even tells you what "absolute value" means.

What else can you do?

We have shown you the basic ideas of looking at specifications of classes and methods. There are a few other things you can do using the icons at the top of the web page. For example, clicking `Index` will open a page that contains an alphabetical listing of all the packages, classes, interfaces, methods, and fields in the API packages.

Experiment with the different features in order to get a good idea what is available in the Java API specs. Use this tool regularly, whenever you need help with some predefined Java class.

We suggest that you browse the spec of class `String`, also in package `java.lang`. This will help you learn about an important class as it gives you some facility with browsing the API package.

Finding API specs on the web

Wrapper classes

An instance of class `Integer` contains a field of type **`int`**. We haven't given the name of the field because we don't know it. But there is a getter method for it, `intValue()`. In fact, one can obtain the **int** value as a primitive value of other types —**`byte`**, **`short`**, and so on— and also as a `String`, using these functions:

```
intValue()      byteValue()      shortValue()      longValue()
floatValue()    doubleValue()
toString()
```

But there is no setter method, so the field can't be changed. We say that it is *immutable*.

Instance function `equals` yields true iff its parameter is an object of class `Integer` and the parameter's wrapped value equals the instance's wrapped value.

```
Integer b= new Integer(5);
b.equals(new Double(5))        is      false
b.equals(new Integer(6))       is      false
b.equals(new Integer(5))       is      true
```

Using wrapper class Integer

`Integer` is called a *wrapper class* for type **`int`**, because an instance wraps, or contains a single integer of type **`int`**, like you wrap a sandwich in saran wrap or cellophane or napkin. We can use an assignment statement to wrap an integer in an instance of `Integer`:

```
Integer x= new Integer(25);
```

So, we have one reason for the existence of type `Integer`:

> Reason for wrapper class `Integer`: to allow us to handle a primitive **`int`** value like an object.

In the next web lecture, we use this technique to put an integer into an instance of class `ArrayList`.

Constants of class Integer

A second reason for class `Integer` is:

> Reason for wrapper class `Integer`: to provide useful constants and functions that deal with **`ints`**.

Thus, class `Integer` provides constants for the minimum and maximum values of type **`int`**. Since these are static fields, one can reference them using the name of the class —we will introduce static fields later.

```
Integer.MIN_VALUE
Integer.MAX_VALUE
```

Static functions of class Integer

Class Integer has a number of static functions that deal with **ints**. For example, one can find the binary, octal, and hexadecimal representations of integers as strings:

```
Integer.toBinaryString(25)     is      "11001"
Integer.toOctalString(25)      is      "31"
Integer.toHexString(25)        is      "19"
```

Finally, Integer has a static function to translate a string that contains an integer into an **`int`**:

```
Integer.parseInt("25")         is      25
```

The argument of a `parseInt` call must contain only digits, possibly with a preceding minus sign. Not even blanks are allowed.

```
Integer.parseInt("25 ")        produces an error
```

Summary

In summary, wrapper class `Integer` allows us to handle an **`int`** value as an object and provides some useful functions that deal with **int** values. If you want to get the minimum or maximum **int** value, or you want to see the binary, octal, or hexadecimal representation of an integer, or you have a string that has to be translated to an integer, then turn to class Integer for help.

Wrapper Classes

Each primitive type has a corresponding wrapper class, which wraps a value of that type and provides constants and functions dealing with that primitive type. We do not go into detail here. Look at the API specs on the web or the ProgramLive CD, page 5-1, for a list of constants and functions for the wrapper classes for the numerical primitive types, which are:

```
byte:   Byte
short:  Short
int:    Integer
long:   Long
float:  Float
double: Double
```

For similar information for these wrapper classes:

```
char:    Character
boolean: Boolean
```

look at the API specs on the web or the ProgramLive CD, page 5-2. Class `Character` has functions for converting between lower-case and upper-case, for testing whether a character is a digit, letter, or whitespace, and so on.

Autoboxing

Up through version 4 of Java, an assignment like

```
Integer b;
b= 25;

int i;
i= b;
```

were illegal, because the types of the variable and expression did not match —one is an **int** and the other is class Integer.

In version 5 of Java, *autoboxing* was introduced, which makes such assignments legal. In the first assignment, `b= 25;` the integer 25 is automatically wrapped in a new instance of class `Integer`; it is as if the programmer had written

```
b=  new Integer(25);
```

In the second assignment, i= b; since an **int** value is needed, the int value is automatically extracted from Integer b; it is as if the programmer had written

```
i= b.intValue();
```

Autoboxing is a convenience for the programmer, allowing them to shorten their code.

Autoboxing is done with *all* primitive types and their corresponding wrapper classes. For example, here is autoboxing of a character:

```
Character c= 'g';
```

Moreover, the autoboxing can be carried out in places other than assignments.

We give the rule for autoboxing for type **int** and leave to you to write the rule for other primitive types.

> **Autoboxing rule for int**: If an expression `e` (say) of type **`int`** appears in a position in which an object of class `Integer` is expected (or possible), the expression **`new`** `Integer(e)` is used in place of `e`.

Here is an example of the use of this rule. The parameter of function isFive,

```
/** = "obj is 5". */
public static boolean isFive(Integer obj) {
    return obj.equals(new Integer(5));
}
```

is of type `Integer`. Yet, in a call of it, we can put a value of type **`int`**.

AutoboxDemo.`isFive(5)`

There is an unboxing rule also —a rule for extracting a value from a wrapper class object. Here is the rule for class Integer; you can generalize it to the other wrapper classes yourself.

> **Autounboxing rule for Integer**: If an expression `e` (say) of type `Integer` appears in a position in which an expression of type **`int`** is expected, then the expression `e.intValue()` is used in place of `e`.

Here is an example of the use of this rule. The parameter of function isFour,

```
/** = "c is 4". */
public static boolean isFour(int c) {
   return c == 4;
}
```

We can call `isFour` using the call AutoboxDemo.`isFour(new Integer(4))`, and the **int** value is extracted from the argument expression.

Class ArrayList

An instance of class java.util.ArrayList contains a list of objects. It may be a list of the courses you have taken, a list of high temperatures for each day in Ithaca in 2006, or a list of objects of totally different classes. There are methods for adding elements to the list, deleting elements, searching for an element, and so on. We illustrate these now, using DrJava as well as the powerpoint window so that you can see what is happening.

Creating an ArrayList and appending elements to it

First, let's import class `java.util.ArrayList` and then create a new `ArrayList` —storing it in `b`. By calling function `size` of the new `ArrayList` object, you can see that it is initially empty.

Let's add a `String` object. As you can see, `add` is a function, which always returns **`true`**. The reason for this is beyond the scope of this course. Of course, you can always put a semicolon after the call to `add` and make a statement out of it, as we do here. If you give function `add` an integer as argument, autoboxing will automatically create a new object of class `Integer` that wraps the integer, and this `Integer` object will be added to the list. By calling function `size`, we see that the size of the list is now 3.

Function toString

The description returned by function `toString` is a list of the objects, separated by commas and delimited by square brackets. Or, we can abbreviate and simply use b, and b.toString will be called. Use this function whenever you want to see what is in an `ArrayList`.

Inserting elements at a specified position

As you can see, the objects in the list are numbered 0, 1, 2, and so on. You can insert an object at any position you want, using the two-parameter function `add`. To show this, let's add an integer 7 at position 1 —note how elements 1 and 2 are shifted to positions 2 and 3. Let's call function `toString` to make sure that this works as we said —yes, it does.

Inserting an object anywhere but at the end takes more time than appending an object to the list, because a number of elements have to be shifted down. The time used is proportional to the number of elements shifted.

Retrieving objects from an ArrayList

Function `get` retrieves the name of the object at the specified position. To show this, let's get the first element, which should be the string "abc".

The elements of the `ArrayList` have apparent class `Object`, and if we assign it to a `String` variable, it will be cast appropriately. If we try to assign it to some other variable, say of class `Integer`, we get a class cast exception. So be careful when retrieving elements from an `ArrayList`.

Changing the element that is in a list

As you can see, list `b` contains four elements: the string "abc" and three wrapped integers. You can change the element at position 2 to the wrapped boolean value **`true`** by calling function `set(2, true)`. As an added benefit, this function call returns the element that was previously at position 2. Let's print `b` again so that you can see that the change actually happened.

Removing an element

Finally, use function call `b.remove(1)` to remove the object at position 1 from the list and return it. This call requires elements numbered 2, 3, …, to be shifted up to positions 1, 2, …, so it takes time proportional to the number of elements moved. Let's make sure it was removed by typing b in the interactions pane.

Summary

The purpose of this lecture was to show you how a list of objects could be maintained in an instance of class `ArrayList`. We have showed you the basic methods. There are others —for example, for searching the list for a value. Peruse the API spec of class `ArrayList` to see what other methods are available.

We have not, and will not, shown applications that use `ArrayList`.

Class ArrayList

Constraining the elements of an ArrayList

You now know that you can create an instance of class java.util.ArrayList and save it using an assignment like

```
ArrayList b= new ArrayList();
```

The elements of this list have the apparent class `Object`. When an element is added to the list, it is automatically promoted to `Object`.

Beginning in Java 1.5, you can constrain the objects of a list by placing a class name, in angular brackets (the less-than and greater-than symbols) after the class name `ArrayList`.

```
b= new ArrayList<Integer>();
```

Now, only objects of class `Integer` can be added to `b`. An attempt to add an object of a class that is not `Integer` or a subclass of `Integer` will be flagged as illegal, and the program won't compile.

You can put any class in place of `Integer`.

The ability to constrain `ArrayLists` in this fashion simplifies some coding and provides more type safety. By using a constrained `ArrayList`, you can reduce the chance of making an error by adding something to an `ArrayList` that does not belong there. Constrain you `ArrayLists` where possible.

Specifications of methods

You know what the first function `mini` does, because its specification, in the comment preceding it, tells you. You don't need the function body —which we show you now.

You have no idea what the second function does, because its specification has not been given. To understand what it does, you would have to look at its body: `Integer.signnum`.

A specification of a method tells us what a call on a method does. The specification should be precise, complete, and as clear as possible. So that you don't have to look at the method body.

In fact, *every* method should have a specification, and you should *never* have to look at the method body when writing a call on method. The method body indicates *how* a task is carried out. When writing a call on a method, we don't want to know that, we want to know *what* the task is. And the spec should tell us.

Writing procedure specifications

We write method specs as *Javadoc comments* immediately preceding the method. Javadoc comments begin with `/**` and end with `*/`. The next lecture will explain why we use Javadoc comments.

Let's begin discussing specs for procedures. There are two points to make here.

```
/** Set the title to t */
public void setTitle(String title){...}
```

1. The spec mentions every parameter by name, and in a way that explains what that parameter is for. If a spec doesn't mention a parameter, the reader may not understand what the parameter is for, and the spec is incomplete.

2. This spec is a *command* to do something. We don't say, "this procedures sets the title to t", or something similar. We say, "Set the title to t." This is because we use it as follows. Suppose we have a call on procedure `setTitle`: `setTitle("I want peace");` . The way to figure out what it means is to replace it by the specification, but substituting the argument for the parameter:

```
Set the title to "I want peace".
```

Since the procedure call is a command to do something, so should be the specification.

Writing function specifications

Since a function returns a value, the spec should be a description of that value. In fact, it should say what a call on the function equals. For example, that's what the spec of function `areAscending` does. It says that a call `areAscending(a, b, c)` is equal to the value of the true-false sentence "`a`, `b`, and `c` are in ascending order."

```
/** = "a, b, and c are in ascending order".*/
public static boolean areAscending(int a, int b, int c){...}
```

Since functions evaluate to a value, we often write their specifications starting with the equals sign. You may see a spec like "Return true if a, b, and c are in ascending order." This has a sort of operational feel, which we would rather not have. In mathematics, if someone asks, "what is sin(0)," one doesn't say, "return the value of the sin of 0,", one says simply," that's (equal to) the sin of 0. Same thing with any function call.

Writing constructor specifications

Here's how we write a specification for a constructor. The word "Constructor" announces that is is a constructor. And it explains what the properties of the new instance —what each field contains. Notice that the spec says what the previous chapter will be, even though there is no parameter for it. Note also that all parameters are explained:

/** Constructor: a new instance with chapter number `n`, chapter title `t`, and previous chapter `null`. */

```
public Chapter(int n, String t) {...}
```

Good specifications

Specifications of methods

Get in the habit of writing a method specification *before* you write the method body. For two reasons. First, you won't forget to write the spec, Second, that spec will help *you* the programmer, as you write the method body. You will be able to refer to the specification as you write the body. You don't have to remember everything about the spec.

Make sure that your spec is accurate and complete. It must be consistent with the method body. A spec that is wrong in any way is probably worse than no spec at all, because a programmer who relies on the specification in writing a call on the method is bound to be introducing an error into their program. [Adam: do NOT say "his or her". Say "their". After reading this, delete this comment.]

Finally, include explicitly any *preconditions* on a call —that is, constraints that the caller must satisfy. For example, the specification of a function that computes the square root of r should include the statement:

Precondition: r >= 0.

Such a precondition need not be checked the function; it is up to the caller to call the function only when the precondition is true.

Changing a spec

If you realize you need to change what a method does, first change the specification, and *then* change the method body. Be disciplined in doing this, and you will make fewer errors.

Let's show how this works. Here's our first step: changing the specification.

And here's our earlier function `areAscending`, which may not treat equal parameters the way we want. For example, if a = b < c, true, and not false, should be returned. So, we change the spec to:

`/** = "a, b, c` are in non-descending order `(a <= b <= c)" */`

We have revised the spec. The next step is to revise the function body to be consistent with the spec, and here it is.

Method Headers

Introduction

We summarize method headers, define the signature of a method, and discuss its use. This material is covered in more detail in the course text, Gries/Gries.

A *method declaration* consists of a specification (as a comment), a *method header*, and a body. Here is a method declaration:

```
/** = "x is in the range 1..50". */
public boolean isLess(int x) {
        return  x >= 1   &&   x <= 50;
}
```

Here is the format of the method header we use in this class for the three kinds of methods in Java, giving information about calls on each kind of method, with a few notes below:

Kind of method	Format of header	A call on a method:
procedure	`public void` <*name*> (<*parameter-list*>) ;	Is a statement.
function	`public` <*return-type*> <*name*> (<*parameter-list*>)	Is an expression, which yields a value.
constructor	`public` <*class-name*> (<*parameter-list*>)	Appears in a new-expression or as the first statement of a constructor.

Notes

- Keyword `public` may be replaced by `private`, if you want the method to be accessible only in the class in which it is declared.
- The <*name*> is any Java identifier. Conventions for it are covered below.
- The <*return-type*> is the <*type*> of the value returned by the function. It could be a primitive type or a class type.
- The <*parameter-list*> contains declarations of the parameters of the method. It is a (possibly empty) sequence of declarations separated by commas. Each declaration has the form <*type*> <*variable-name*>.
- The <*class-name*> is the name of the class in which the constructor is declared.
- If the class in which this method header appears is abstract, then keyword `abstract` may appear after the access modifier (`public` or `private`). This is explained in another section.
- For a function or procedure, place keyword `static` after the access modifier if the method does not refer directly to an instance variable or instance method of the class. Then, there will be only one copy of the method, and it will be placed in the file-drawer of the class.

Method names

A method name is often written as a sequence of one or more words, like `toString`, `setName`, and `isLess`. The first letter of each word, *except the first*, is capitalized.

A procedure name is usually a command to do something, e.g. `setName`, `printArea`, `fixName`.

A function name is usually one of:

- a noun phrase that describes the value returned, e.g. `sin`, `sqrt`, `smallestName`;
- the name of an algorithm, e.g. `quicksort`, `selectionSort`, `binarySearch`;
- for a boolean function, a true-false phrase that indicates its value, e.g. `isLess`, `areRed`;
- for a getter method, the name of a field preceded by `get`, e.g. `getTitle`.

Method signature

The signature of a method consists of its name and the types of its parameters (enclosed in parentheses and separated by commas). For example, the signature of the method declared near the beginning of this webpage is

```
isLess(int)
```

In Java, the methods in a class must have different signatures. This means, for example, that one cannot have both a procedure and a function with name `isLess` and one `int` parameter. Java constrains methods in this fashion so that, for any method call, there is no ambiguity concerning which method is being called.

Function and Procedure Calls

A function call has the form

<*function-name*> (<*arguments*>) // this is an expression

and a procedure call has the form

<*procedure-name*> (<*arguments*>) ; // this is a statement

where <*arguments*> is a sequence of expressions separated by commas.

The method call may have to be preceded by an expression that indicates where the method is. For example,

- `Math.sqrt(7)` calls static function `sqrt` in class `Math`.
- `s.length()` calls function `length` of the object whose name is in variable `s`.
- `s.substring(3,5).length()` first calls the two-parameter function `substring` of the object whose name is in `s` to produce a second `String` object and then calls function `length` of that second object.

The number of arguments must equal the number of parameters of the method being called, and each argument type must be the same as or narrower than the corresponding parameter type.

A method call is executed as follows —this is explained in more detail in a later lecture:

1. Create the parameters and local variables of the method being called.
2. Assign the values of the arguments to the corresponding parameters.
3. Execute the method body.
4. Erase the parameters and local variables —and, if this is a function, return the value of the expression of the return-statement that terminated the method body.

From the above, you can see that the arguments are evaluated once and their values are stored in the parameters. In the method body, storing a value in a parameter has no effect on the corresponding argument.

Local variables

A *local variable* is a variable that is declared within a method body. The program you see has two different local variables, both named `temp`.

The syntax of a local variable declaration is:

<type> <variable-name> ;

and it can be an initializing declaration:

<type> <variable-name>= <expression>;

The initializing declaration allows us to write a sequence of statements to swap two variables as follows:

```
int temp= x;  x= z;  z=  temp;
```

We limit our remarks on local variables to the three issues that you see on your monitor:

1. When a local variable is created and destroyed.
2. The scope of a local variable.
3. Naming a local variable.

When a local variable is created and destroyed

Here is a call on method `m`, with three arguments. In executing this call, all local variables are created before execution of the method body. Note that there are two declarations for variable `temp`, They are different variables, and that is why you see two variables with that name. Second, the local variables remain in existence as long as the method body is being executed. Third, destroyed when execution of the method body terminates.

Knowing when local variables are created —before execution of the method body begins— will help you answer questions about local variables that will arise later on. So please memorize when local variables are created.

The scope of a local variable

The scope of a local variable is the place in which it can be referenced. This includes all statements in the block in which it is declared, starting with the statement following its declaration and going to the end of the block.

For example, in this function, the scope of `s` begins at the declaration of `k` and ends with the return statement. And the scope of `k` begins with the for-loop and ends with the return statement. This means that one cannot refer to `k` in a statement before its declaration.

Note that the scope of a variable declared in the initialization of a for-loop consists of the rest of the for-loop, but not the statement following the for-loop. In the example shown here, `k` is declared within the initialization of the loop, so it cannot be used in following the loop.

Guidelines for naming a local variable

Now let's talk about local-variable names and parameter names. Generally, methods in OO programming are usually short. In fact, they generally take less than a page, and almost always less than two pages.

The brevity of a method means that parameter declarations as well as the specification of the method, which describes what each parameter is for, can always be seen when reading the method body. Therefore, there is no need for long parameter names.

One will sometimes see a method like this with long parameter names —the programmer is trying to put the meaning of the parameters into their names. But instead of clarifying and simplifying, this change has complicated. How much harder the program is to read! Doesn't it look awkward? And there is no need for this complication; the short names, with no meaning, are OK because the specification and the other comments gives all the meaning, and they are readily seen because the method is short.

In the same way, if local variables are sufficiently described in comments, and if the method body is short, short names are to be preferred for local variables.

Declaring local variables where they belong, logically speaking

We discuss the placement of local variable declarations. Generally, the declarations should go where they belong, logically speaking, and this usually means placing them as close to their first use as possible. Let's look at an example that illustrates this.

Here is a procedure to sort an array segment `b[0..n]` using an algorithm called *selection sort*. We have not discussed arrays in this course, but you have seen them in another programming language. In the first parameter declaration, you see how to declare `b` to be a one-dimensional array.

Just after the declaration of `temp and` j is a comment that states the invariant of the main loop —this is simply a picture that holds before and after each iteration of the loop, and we display the information as a picture down below. The picture says that the segment `b[0..k-1]` is sorted and that all of its elements are at most those in the second segment, that's what we mean by the signs <= and >=.

Take a look at the repetend of the outer loop. It sets `j` to the index of the minimum value in `b[k..n]` and then swaps `b[j]` and `b[k]`, so that `b[k]` contains the smallest value in the second segment `b[k..n]`. Since `b[k]` is the smallest value of that segment, and it is at least all the elements in the first segment, incrementing `k` in this segment keeps the picture true.

Now, why are `temp` and j declared at the beginning? What is their meaning? When you first read the procedure body, aren't you confused? In fact, these variables have no meaning at this point, and their appearance at the beginning of the body is logically incorrect. Their appearance at the beginning gives the impression that they have meaning from iteration to iteration, which they don't. The declarations don't belong at the beginning.

Consider variable `temp`. It is used only in swapping `b[j]` and `b[k]`, so `temp` can be declared in the first assignment.

Consider variable `j`. Where is it used? Only in the repetend of the outer loop: it will contain the index of the minimum of `b[k..n],` and then it will participate in a swap. So we can declare `j` at its first assignment.

Note that `temp` is declared within the *statement-comment* "Swap `b[j]` and `b[k]`", because that is where it is used, that is where it belongs. And, the declaration is as close to its first use as possible.

Similarly, `j` is declared where it belongs; it is a local variable of the repetend of the outer loop, and it is as close to its first use as possible.

Are you worried about making the procedure less efficient because the declaration is "processed" many times? Don't be worried. Remember, a local variable is created before execution of the method body, just once. So we have not made the procedure less efficient. Moreover, the procedure is easier to understand.

Conditional Statements and Loops

Introduction

Every procedural programming language has conditional statements (if-statements) and iterative statements (loops). You should be familiar with the concepts —unless your only previous programming language was a purely functional language. Therefore, our purpose here is to summarize the Java versions of these statements and point you to the places in the text and the CD *ProgramLive* for longer discussions of these statements.

1. The block.
2. if-statement.
3. if-else statement.
4. while-loop.
5. for-loop.

The Block

The block has the form

{ <*sequence of statements and local-variable declarations*> }

Its purpose is to aggregate (collect together) the sequence of statements and declarations into a single statement. It used below in writing conditional and iterative statements.

For more information, see Gries/Gries, p. 69, and ProgramLive, p. 1-4

---

if-statement

The if-statement has the form

if (<*condition*>) <*statement*>

The <*condition*> is a boolean expression. We generally make the <*statement*> be a block.

The following example shows the conventions used in this course for indenting and the placement of the opening brace of the <*statement*>.

```
if (x < y) {
        // Swap x and y
        int temp= x;
        x= y;
        y= temp;
}
```

To execute the if-statement, evaluation the <*condition*> and, if it is true, execute the <*statement*>.

For more information, see Gries/Gries, p. 69, and ProgramLive, p. 1-4.

---

if-else statement

The if-else statement has the form

```
if ( <condition> ) <statement1> else <statement2>
```

The <*condition*> is a boolean expression. We generally make <*statement1*> and <*statement2*> be blocks.

The following example shows the conventions used in this course for indenting and the placement of the opening braces of the statements.

```
// Put the smaller of x and y in z
if (x <= y) {
        z= x;
} else {
        z= y;
}
```

or

```
// Put the smaller of x and y in z
if (x <= y) {
        z= x;
}
else {
        z= y;
}
```

To execute the if-else statement, evaluation the <*condition*>; if it is true, execute <*statement1*>; if it is false, execute <*statement2*>.

For more information, see Gries/Gries, p. 69 and ProgramLive, p. 1-4

---

while-loop

The while-loop has the form

```
while ( <condition> ) <repetend>
```

where the <*condition*> is a boolean expression and the <*repetend*> is any statement. (*Repetend* means *the thing to be repeated*.)

We generally make the <*repetend*> be a block.

The following example shows the conventions used in this course for indenting and the placement of the opening brace of the <*repetend*>.

```
// Precondition: n >= 1
// Set x to the largest power of 2 that is at most n
```

```
x= 1;
// invariant: x is a power of 2 and x <= n
while (x <= n) {
        x= 2 * x;
}
```

To execute the while-loop, do the following:

1. Evaluate the <*condition*>; if it is false, terminate execution of the loop.
2. Execute the <*repetend*>; then, repeat from step 1.

The first execution of the <*repetend*> is called iteration 0; the second, iteration 1; and so on.

For more information, see Gries/Gries, p. 233, and ProgramLive, p. 7-1.

---

for-loop

We don't describe everything about the for-loop but just its use with a single "control variable" that is initialized at the beginning and incremented every iteration of the for-loop. (An iteration is one execution of the repetend —see below.) The for-loop has the form

for (<*initialization*> ; <*condition*> ; <*increment*>) <*repetend*>

where the <*initialization*> is an assignment or an initializing declaration, the <*condition*> is a boolean expression, the <*increment*> is an assignment, and the <*repetend*> is any statement. (*Repetend* means *the thing to be repeated*.)

We generally make the <*repetend*> be a block.

The following example shows the conventions used in this course for indenting and the placement of the opening brace of the <*repetend*>. Note that a range `m..m` has one value, `m`. But the range `m..m-1` has no values and thus has the sum 0. Finally, if `m= n+1` in this example, then 0 iterations (executions of the repetend) of the loop are performed.

```
// Precondition: m <= n+1
// Set x to the sum of the value in the range m..n
x= 0;
// invariant: x is the sum of the values in the range m..k-1
for (int k= m; k <= n; k= k+1) {
        x= x+k;
}
```

Execution of the for-loop is equivalent to execution of the following translation into a while-loop, with one exception. If the initialization is an initializing declaration, the scope of the control variable is the for-loop itself, and the control variable cannot be referenced in statements after the loop. But in the while-loop version, the scope includes the statements following the loop.

<*initialization*> ;
while (<*condition*>) {

```
        <repetend>
        <increment>
    }
```

For more information, see Gries/Gries, p.78 (a quick introduction) and p. 253, as well as ProgramLive, p. 7-4.

Arrays

Introduction

Java, like most programming languages, has arrays —lists of elements of a certain type, where the size of a list is fixed when the array is first created. In Java, arrays are objects.

We provide a very brief summary of one- and two-dimensional arrays. Turn to Chapter 8, pp. 271–297, of Gries/Gries for complete discussion of one-dimensional arrays. Read Chapter 9, pp. 301–311 to see how a two-dimensional array is handled as an array of arrays.

Declaration of one-dimensional array variables

Use

```
int[] b;
JFrame[] j;
```

to declare a variable `b` that can contain an object that is an array of `int`s and a variable `j` that can contain an object that is an array of `JFrame`s. Arrays are objects, and variables `b` and `j` can contain the names of array objects. Initially, `b` and `j` contain `null`.

Note that `int[]` and `JFrame[]` are types. We read these as type "**int** array" and type "JFrame array".

Creation of an array

Evaluation of the new-expression

`new` <*type*> `[` <*integer-expression*> `]`

creates an array that contains <*integer-expression*> elements, all of type <*type*>. So, below we create an array of 10 `int`s and an array of 20 `JFrame`s and store them in `b` and `j`:

```
b= new int[10];          // Initally, each array element contains 0.
j= new JFrame[20];       // Initially each array element contains null.
```

The number of elements in the arrays are given by `b.length` and `j.length`.

Referencing and changing array elements

Array indexing starts at `0`; e.g. the elements of array `b` are referenced using `b[0]`, `b[1]`, ..., `b[b.length-1]`. The following assignment adds `1` to `b[k]`:

```
b[k]= b[k] + 1;
```

and the following procedure call sets the title of `j[3]`:

```
j[3].setTitle("Element number 3");
```

Two-dimensional arrays

A two-dimensional `int` array has type `int[][]`. So, to declare an array variable of this type, use:

```
int[][] c;
```

To create a two-dimensional 20-row by 30-column `int` array and store it (i.e. its name) in `c`, write:

```
c= new int[20][30];
```

The number of rows, `20`, is given by `c.length`; the number of columns in row `3` by `c[3].length`.

To add `1` to the array element in row `3` column `5` of `c`, write:

```
c[3][5]= c[3][5] + 1;
```

Return Statements

Introduction

Execution of a return statement terminates execution of the method body and, hence, of the method call. The return statement in the body of a function differs from the return statement in a procedure or constructor. See below.

The procedure and constructor bodies

When a procedure or constructor is called, the statements in its body are executed one at a time, in the order in which they appear. During this execution, execution of the statement:

```
return;
```

terminates execution of the body, and thus of the procedure or constructor call —no further statement in the body is executed, once a return-statement is executed. A return-statement need not be included in the body of a procedure or constructor, but sometimes a body may be easier to write using a return-statement.

The function body

A function call yields a value. Therefore, execution of a function body must terminate by executing a return-statement of the form

`return` <*expression*> ;

where the type of the <expression> is the same as (or narrower than) the type of the result of the function. Execution of such a return-statement terminates the function body, and, thus, the function call, and yields the value of the <*expression*> as the result of the call.

Since execution of a function body must terminate with execution of a return-statement, the last statement of a function body is usually a return-statement. However, return-statements may appear in other places as well.

Static methods and variables

Setting the scene

Here's a version of class `Chapter`, with three fields: the chapter number, the chapter title, and the previous chapter. There are the usual constructor and getter methods and setter methods —we don't show them all.

Here is the file drawer for class `Chapter`, showing two objects of the class. Again, not all methods are shown in the objects.

We also declare a function `isBefore`, which tells whether the object in which it appears comes before chapter `c`. And the new function appears in all objects of class `Chapter`. There is an important reason for the function to be in each object. Each method body contains a reference to field `number`. In object `a0`, the reference to `number` accesses field `number` in `a0`. In `a1`, the reference to `number` accesses field `number` in `a1`. This is a consequence of the inside-out rule, which we discuss in detail in the next blecture.

A function that references no components

Here's another function with the same intent, to see whether one chapter comes before another one. In this function, both chapters are given as parameters. We now have two functions with the same name; the name `isBefore` is overloaded.

This function appears in every object of the class. However, it doesn't reference any fields and doesn't call any methods in the class. So why should this function appear in every object? There is no need for that.

We can delete this function from the objects by giving it attribute **`static`**. Now, a *single* copy of it is in the class file drawer. That's right:

> **Rule**: A method that is declared static appears in the file drawer —and *not* in each object.
> There is only one copy of the method.

We can't make the first function `isBefore` static because its reference to number is now illegal —there is no variable `number` in sight. Here, we'll show you how Java complains when we make that function static and attempt to compile.

So, we take off the static attribute. And, we have the following guideline:

> **Guideline**: Make a function or procedure static if its body does not refer to a field or instance method of the class.

Suppose you want to use only one function `isBefore`. Should you use the nonstatic version or the static version? We can't really answer that question now. The answer depends on the class being defined and how one is expected to use the class, and this is not the topic of this discussion. Here, we focus only on the mechanics of defining static methods and the consequences.

Class variables

One can also declare static variables in a class. To show this, we declare a variable `numberOfChaps`, which is to contain the number of objects of class `Chapter` that have been created thus far.

```
// no. of chapters created so far.
private static int numberOfChaps= 0;
```

Such a variable appears directly in the file drawer. And there is only one copy of it. It is created just before *any* static component is referenced or the first instance of the class is created. It is initialized either to the default value for its type or to the value given in an initializing declaration.

A static variable is also called a *class variable*, because it belongs to the class, or file drawer, and not with each object of the class.

When should a variable be made static? When only one copy of the variable is needed, for example, when it accumulates information about all objects in some way, or communicates information about all objects in some way. Class variable `numberOfChaps` is an example of this. It wouldn't do to have this variable in all objects.

Maintaining class variable numberOfChaps

Class variable `numberOfChaps` is supposed to contain the number of objects of class `Chapter` that have been created. The simplest way to maintain the variable is to increase it by 1 whenever an object is created, and

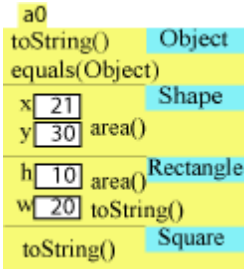
that indicates that it should be increased in the constructor —because the constructor is called whenever a new object is created.

```
numberOfChaps= numberOfChaps + 1;
```

The Bottom-Up Rule

The bottom-up rule defines how to find a variable, or a method with a matching signature, in an object:

> **Bottom-up rule**: When looking in an object for a variable, or for a method with a particular signature, start from the bottom of the object and look up upward until it is found (or until it is determined that it is not present).



We use object `a0`, to the right, to illustrate. It has four partitions. From it, we see that `Square` is a subclass of `Rectangle`, which is a subclass of `Shape`, which is a subclass of `Object`. The expression

```
a0.h
```

refers to variable `h` in partition `Rectangle`, since that is the first (and only) `h` encountered when searching from the bottom of the object to the top.

The call `a0.area()` calls the method defined in partition `Rectangle`, while `a0.toString()` calls the method in partition `Square`, since these are the first matching methods encountered when searching from bottom to top.

For methods, the bottom-up rule always gives us the *overriding* method, if a method is inherited in a subclass.

In this course, we refrain from redefining an inherited variable in a subclass and don't explain what happens in this case. Redefining an inherited variable is rarely done, because there is little use for it. Therefore, for a variable, the bottom-up rule finds the *only* variable of the given name in the object.

The inside-out rule

Caveat about respecting privacy

We show you a general rule for determining the variable declaration associated with a reference to a variable. The issue is independent of whether the variable reference is actually legal. For example, in method `m` in this `Section` object, the reference to variable `title` is associated with field `title` inherited from superclass `Chapter`. But if field `title` is declared with access modifier **`private`**, the reference is illegal, and probably function `getTitle` should be called instead.

The inside-out rule for variables

Each variable name in a program refers to a variable that is declared somewhere within the program. Most programming languages use some form of an inside-out rule to determine the declaration to which a variable-name refers.

For example, the variable might be declared in the body of a loop in which the variable reference appears; if not there, in a surrounding if-statement; if not there, in the body of the method; if not there, it might be a parameter; if not there, it might be a field of the object in which the method is declared; and finally, if not there, it might be a static variable and thus in the file drawer.

Thus, the search is made in the inner scope and expands to increasingly surrounding scopes. That is,

> **Inside-out rule**: To find the declaration corresponding to the use of a variable, look in enclosing scopes from inside-out, until it is found.

In DrJava, we see a class that illustrates this. You see local variables declared in the loop body, the then-part of the conditional statement, and the method body. You see a parameter declaration. You see a field. And you see a static variable.

This inside-out rule, then is the basic mechanism for determining the declaration of a variable that a variable reference references. It is very simple.

The inside-out rule for method calls

The inside-out rule for method calls is similar. There are two differences. First, methods appear only in objects and in the file drawer, so the search is simplified.

For a method in an object, one need look only in that object and then in the surrounding file drawer. For example, method `toString` in object `a0` calls `getTitle`; its declaration is found in object `a0` itself. Method `toString` also calls `m` with an **`int`** argument. That method is not in `a0`, but it is found in the enclosing file drawer.

For a static method, which is in the file drawer, one can look only in the file drawer, because there are no enclosing scopes. For example, the method `m` with a **`boolean`** parameter calls method `m` with an **`int`** argument, and that method `m` is also in the file drawer.

Here is the other difference between searching for a variable and searching for a method. When searching for a method, a method is needed not with the right name but with the right signature —the name together with parameter types. So, in looking for a method declaration for the call m(number), a search is made for a method with a single parameter whose type is as least as wide as **int**.

Looking into a folder or a file drawer

Here are two file drawers, one for class `Chapter`, and the other for class `C`. By the inside-out rule, from the body of method `toString`, all methods and fields of `a0` as well as the static methods and variables in the file drawer can be referenced. From the body of the two methods `m`, the static methods and variables in file drawer `Chapter` can be referenced, but not the components of object `a0`. The inside-out rule doesn't allow looking inward, only outward.

From method `p` in object `a1`, none of the items in `Chapter`'s file drawer can be referenced directly.

However, within method `p`, we can reference field c1, which contains the name a0. Therefore, all components in a0 can be referenced, as well as the static components in Chapter's file drawer, using references like

The inside-out rule

```
c1.title     c1.getTitle()     c1.m(3)    c1.m(true)
```

Thus, the name of an object gives access not only to the components of the object but also, by the inside-out rule, to the static components in the file drawer.

Just as the name of the object is used as a gateway to that object, the name of a class is used as a gateway to the static components in the class file drawer. For example, in the body of method `p` in class `C`, we can call method a method m of in `Chapter`'s file drawer by preceding the call with the name of the class, `Chapter`:

```
Chapter.m(n)
```

A static component of a class, then, can be referenced from anywhere simply by prepending the name of the class (followed by a period, of course) to the reference.

Note that in method `p`, we can use either `c1.m(n)` or `Chapter.m(n)` to call the first method `m`. The use of the name of the `Chapter` is preferred.

The class invariant

Each instance of this class describes a time of day in some part of the world. The time itself is in seconds. Field `zone` contains a three-letter designation of the time zone for the current time, like `"GMT"` for Greenwich Mean Time, or `"EST"` for Eastern Standard Time. And field `ampm` indicates whether this time should be displayed in AM-PM time or in 24-hour time.

The class has a bunch of methods: three constructors, one of which allows the time to be given not in seconds but in hours, minutes, and seconds; getter methods; one setter method; and function `toString`.

It also has a function that yields an object with the same time but in zone GMT. For example, if we call this function in an object that has a time of 8 hours in EST, it produces a new object that has the same time in GMT —the time is 13 hours, because EST, which is New York time, is 5 hours behind GMT time, which is London time. But if object `a0` has time 23 hours, New York time, then the new object will have time 28 hours in GMT. This kind of conversion might also end up with field `time` containing a negative value.

Now, consider writing the constructor that is given the time, in seconds, and the zone. In order to ensure that objects have no mistakes in them, we require that zone z be a legal zone. But what is a legal zone? How many different ones are there? Which ones will this program handle? We have not been given this information.

There *should* have been a comment on the declaration that defines the zones that can be used.

Next, the constructor should make sure that the time, s seconds, is appropriate as well. But what is appropriate? Are there any constraints on it? We can't tell. There *should* have been a comment on the declarations that explains what field time is and gives constraints on it. The main property is that in *some* time zone, the time is in the range 0 to 24 hours. And this helps us make the specification of the constructor more precise.

Finally, there should be a comment explaining what field `ampm` is for.

The class invariant

These comments together constitute thee *class invariant*. This is a description of the fields of an instance. It must include all constraints on them. For example, in this case, only certain values are allowed for the zone.

The class invariant is used as follows. Every method in an object can assume that the class invariant is true when the method is called, and the class invariant must be true when execution of the method body ends. The programmer can rely on all the constraints when writing the method body. In return, the programmer is responsible for ensuring that the class invariant is true when the body terminates.

It is your duty as programmer to write the class invariant when you declare the fields. And the more careful and clear and precise and thorough you are in writing the class invariant, the easier time *you* will have in writing the methods. In fact, you, the programmer, are the one who will benefit the most from writing the class invariant early in the game. Let me show you why.

Consider writing the constructor. From the specification, the body has to check whether the zone is legal. But what is a legal zone? Well, just look at the class invariant.

In fact, the class invariant should be looked at continually when writing almost every method. First, knowing the constraints on the fields generally makes writing the body easier. Second, continual referral to the class invariant will help you ensure that the constraints are true when the method body ends.

Position of the class invariant

In this blecture, we placed the class invariant as a single comment before the declarations. As an alternative, parts of the class invariant may be interspersed with the declarations and may appear as comments to the right of a declaration. The important point is not the particular format used but the actual writing of the class invariant.

Calling constructors from constructors

Calling another constructor of in the same instance

Class `Shape` contains the coordinates of the top-left position of the bounding box of some shape. `Shape` meant to be subclassed in different ways to describe different shapes.

This class has *two* constructors. The first allows the user to give the position of the bounding box. The second uses the default bounding-box position (100, 100). One often writes several constructors, to provide flexibility for the user.

The constructor with no parameters directly assigns values to the fields. Instead, it should be possible to call the other constructor, say using

```
Shape(100, 100);
```

However, as you can see, Java does not allow this call. The Java convention to call another constructor in this class is to use keyword **`this`** instead of the class name:

```
this(100, 100);
```

The use of a constructor call in this instance does not save much, since there are only two fields to initialize. But in other cases, where there are more fields or whether a great deal of calculation has to go on in to initialize the fields, the use of a constructor call can save time and effort. Using a constructor call here follows a principle that has been mentioned several times elsewhere:

> **Principle**: Always use previously written methods, in order to save time and effort — both in writing a program and in testing and debugging it.

A constructor call must be first

The call of another constructor from within a constructor must be the *first* statement of the constructor. We can illustrate this by inserting an assignment as the first statement of the constructor, before the constructor call. And compile. There, see the error message that is produced?

Calling a constructor in the super class

Now let's write class `Circle`. We have declared the field, stubbed in the constructor, and written the getter function. Notice the specification of the getter function. It doesn't say "= field radius". The user does not know what the field is named. The spec is written in terms of the meaning of the class and not its implementation. It says simply, hey, get me the radius of the circle.

The constructor has to initialize fields `x`, `y`, and `radius`. It makes sense to follow the principle that:

> **Principle**: When initializing fields of an object during execution of a constructor, the inherited fields should be initialized first.

Since the inherited fields are private, and since there are not setter procedures for them, the inherited fields have to be initialized using a call on a constructor in the superclass. We can't use

```
Shape(x, y);
```

We can see that the program with this statement in it won't compile. Instead of the name `Shape`, use keyword super:

```
super(x, y);
```

And that's how you call a superclass constructor.

Java almost enforces the principal that inherited fields should be initialized first:

Principal: Initalize inherited fields before initializing newly declare fields

You can see this by putting the superclass call after the assignment to field `radius` and compiling. See? There is an error. *A call on a constructor cannot appear anywhere but as the first statement of the constructor body*.

The default constructor

The default constructor

If a constructor is not declared in a class `C`, then Java provides a default constructor. This constructor simply calls the superclass constructor that has no parameters.

```
public C() {
    super();
}
```

But if a constructor *is* declared in a class, then the default constructor is not included, unless you declare it yourself.

We verify this using the class `C` that appears in DrJava. `C` does not explicitly declare a constructor, but we can still create an instance using **`new`** `C()`. However, if we declare a constructor, say to store something in field `x`, and compile, and try to evaluate the same expression **`new`** `C()`, we get a "No Such Method Exception". The method does not exist.

A second point. If a constructor body does not begin with a constructor call, Java inserts the call

super();

Let's illustrate this point. The body of the constructor in class Circle does nothing —it's wrong. Let's create a new instance of Circle. It works. Now let's turn to class Shape and, for the moment, remove the constructor with no parameters. And compile.

Look, now it says that on line 15 of class Circle, which is the beginning of the body of Circle's constructor, a symbol cannot be found: constructor Shape(). So this indicates that on line 15, in the beginning of the constructor body, the call super(); is illegal because a constructor without parameters does not exist in class Shape!

Let's put the constructor back in and recompile. Now, it compiles.

The first time see an error because there is no constructor with 0 parameters, you will be confused. I suggest that you watch this complete lecture again, so that perhaps you will remember about the default constructor — what it looks like and when it is inserted automatically and when it isn't.

Apparent and real classes

The purpose of any terminology is to facilitate discussions of issues and clarify concepts. Without the terminology, discussions may be long and laborious and confusing. With the terminology, the issues and concepts should be explained and discussed in shorter time. In this blecture, we introduce two new words: the *apparent class* and *real class* of a variable or expression. Please make sure you understand this new terminology.

The apparent class of a variable

Here are basic declarations of three variables: `ob`, `sp`, and `ci`.

```
Object ob;   Shape sp;   Circle ci;
```

And here are the variables themselves. We have annotated them with their types. They all contain the name of the same object, of class Circle.

In this situation, we say that the *apparent class* of `ob` is `Object`, the *apparent class* of `sp` is `Shape`, and the apparent class of `ci` is `Circle`:

Definition: The *apparent class of a variable* is the class with which it is declared.

Why do we use this name? *Apparent* means clearly *seen or understood*, *appearing to show particular qualities or attributes that may not be genuine*. Apparently, the variable contains an object of the given class.

The apparent class is a *syntactic property*. You can tell from the program itself, without executing it, what the apparent class of a variable is. And the apparent class determines what can and cannot be referenced.

In the situation shown here, since `ob`'s apparent class is `Object`, only fields and methods that are defined in class `Object` can be referenced using `ob`. Those that cannot be referenced have been grayed out. Since `sp`'s apparent type is `Shape`, only fields and methods that are defined or inherited in class `Shape` can be referenced using `sp`. Those that cannot be referenced have been grayed out. And, since `ci`'s apparent class is `Circle`, only fields and methods that are defined or inherited in class `Circle` can be referenced using `ci`.

In summary, the rule is:

> Rule: For a variable `x` of some class-type `C`, the legal references of the form `x`.*variable* or `x`.*method-call* are to variables and methods defined in or inherited by class `C`.

The real class of a variable

The apparent class is what the object in a variable appears to be from its declaration; the real class is what the object in the variable really is. At the moment, the apparent class of `ob` is `Object`, but the real class of `ob` is `Circle`, because `ob` contains the name of an object that is of class `Circle`.

This real class can change during execution, whenever an assignment to the variable is executed.

The real class determines which method is actually called. For example, in the situation given here, `ob.toString()` is legal, because `toString` is defined in class `Object`. However, it calls the overriding `toString` function that is declared in class `Circle` —this is a consequence of the bottom-up rule for finding the declaration in an object corresponding to a variable or method.

The fact that the overriding `toString` function is called may seem strange at first. But it is an important aspect of OO programming. Here's how to see that it makes sense. When the call `ob.toString()` is evaluated, we would like as much information as possible about the object. If function `toString` in the `Object` partition were called, all we would get is the name of the object! But, since the `toString` function in partition `Circle` is called, we get the maximum amount of information.

Function equals

Equals in class Object

As you know, function `equals`, defined in class object, tests whether two objects are indeed the same object. It can be written as follows:

```
/**  = "This object and object ob are the same object". *
public boolean equals(Object ob) {
    return this == ob;
}
```

For example, for two variables c1 and c2 that you see here, `c1.equals(c1)` is true, and `c1.equals(c2)` is false.

Each class C (say) can override function equals. Here is its general specification:

```
/** = "Object ob is of class C and equals this object */
public boolean equals(Object ob)
```

There are two important points to note here. First, in order to override function equals, the parameter must have type `Object`. Second, the specification says that `ob` is of class C, so this property must be checked in the method body. For example, suppose `c1` contains an object of class C. Then, the call

```
c1.equals(new Integer(5))
```

must yield **`false`**, because the class of the argument is not C.

Function equals should be an equality relation

Now, the writer of function equals gets to decide what "object `ob` equals this object" means. The convention is that this is a real equality —provided `c1`, `c2`, and `c3` are objects of the same class, three properties hold:

Reflexive: `c1.equals(c1)` is true.

Symmetric: `c1.equals(c2)` and `c2.equals(c1)` yield the same value.

Transitive: If `c1.equals(c2)` and `c2.equals(c3)` are true, then so is `c1.equals(c3)`

Now, if you haven't been told that equality has these properties, don't worry. You don't have to check that these properties hold for your function `equals`. If you just define `equals` using common sense, then it should have these properties.

Function `equals` generally checks for equality of some or all fields of the two objects, but it should be specified in an abstract way, in terms of the meaning of the class and not the implementation with fields. For example, equality on `String` objects is defined by

The two strings (sequences of characters) contained in the objects are equal

We don't know what fields are used in class `String`; we know only that, somehow, each instance contains a sequence of characters, and the equality of these sequences is being tested in function equality.

So, write your specification in terms of the meaning of the class and not in terms of its fields.

An example of writing a function equals

We write a function `equals` for class `Chapter`, where a chapter has a number, a title, and a previous chapter. Let's write it step by step. First the specification. Now the header. And we stub in a return statement so that we can compile.

We write the function body. Look at the spec; the first thing it says is that ob must have class `Chapter`. So, let's put in an if-statement to check it and return false if it is not a `Chapter`. Note that if `ob` is false, **`instanceof`** will evaluate to **`false`**, so that the function returns **false**.

Do you see how we refer to the method specification continually when writing the method body? If we hadn't written the spec first, we would not have been able to do this so easily and, quite likely, we would have made a mistake. Write method specs before writing method bodies.

Function equals

Now, we know that `c` is a `Chapter`, so let's define a local variable, cast `ob` to `Chapter`, and store the result is `c`. This is so that the fields of instance `Chapter`, or `c`, can be referenced.

Now comes a single return statement that tests each of the required items. We begin with the chapter number —for emphasis here, we put in keyword **`this`**, although it is not necessary because field number is accessible anyway. Since field number has type **`int`**, we use equality ==. Now comes the chapter title. Since it is a `String`, we use function `equals` class `String` —using ==would be a mistake. Finally, we have the previous chapter. Note that we use equality ==, and not function `equals`! This is important. We want to test whether the two previous objects are the *same* object, not that they are equal in terms of having equal values in the objects.

And that's it! It's rather simple. All `equals` functions will have this form. First, make sure the parameter has the right class. Then, cast it to that class. Finally, make the equality test.

```
/** = "ob is of class Chapter and has the same
     chapter number, chapter title, and
     previous chapter as this object". */
public boolean equals(Object ob) {
    if (!(ob instanceof Chapter))
        return false;

    Chapter c= (Chapter) ob;

    return this.number == c.number &&
           this.equals(c.title) &&
           this.prev == c.prev;
}
```

Notes on Java Programming Style

by David Gries

A goal of CS 1110 is for you to learn to write programs that are not only correct but also understandable. These guidelines should help you toward that goal. We ask that you follow these guidelines when writing programs in this course. They will give you a good basis for developing a style of your own as you become a more experienced programmer.

This handout includes guidelines for Java constructs that will be covered later in the semester. Skim these sections now and read them more carefully later when the topics are discussed.

Table of Contents

- Reasons for a Disciplined Programming Style
- Naming Conventions:
  - Package Names
  - Class Names
  - Method Names
  - (Method) Paramater Names
  - Field Names
  - Local Variable Names
  - Accessor Names
  - Constant Names
- Indentation
- Statement-Comments
- Method Specifications
- Class Specifications
- Variable Definitions
- Acknowledgments

Reasons for a Disciplined Programming Style

Your computer program should be readable, because if it is not readable, the chances of it being correct are slim. Moreover, if your program is unreadable, it will be difficult and time-consuming to find and correct the errors in it. Therefore,

> **The major reason for using a disciplined style of programming is that it will save you time whenever you have to read your program or debug it.**

Your program should be readable by others, not just you. Consider a program you will write in this course. A grader who has trouble understanding it is not likely to give you good a grade. The more understandable, the simpler your program appears to a grader, the better the grade they will be willing to give you.

Outside this course, making programs readable by others becomes even more important. Most programs live a long time and require "maintenance" --changes to adapt to new and different requirements, upgrades in other software, new hardware, etc. Further, the author of the program is quite likely not going to be around when the maintenance is required; someone else must read the program and understand it enough to update it successfully.

Even the programs you write for yourself should be readable; if not, four weeks after finishing it you will not remember it enough to make changes simply.

Thus, simply for your own sake and for the sake of others, it makes sense to develop programming habits that lend themselves to writing readable, understandable, and correct, programs.

Part of these habits concern simple, syntactical measures like indenting program parts properly and using a few conventions for names of variables, methods, etc. The more important part concerns recording enough information in comments for the reader to understand how a program is designed and why.

A computer program is the result many design decisions. These decisions --why this variable was introduced, what that function does, etc.-- are often not reflected in the final Java code, which consists of low-level, detailed declarations and statements. However, the higher-level design *must* be understood if a programmer is to modify the program successfully. Trying to understand decisions that are not recorded in comments in the code is tedious, error-prone, and aggravating --but all too common.

So, it will be to your advantage to instill in yourself some disciplined programming habits, right from the beginning, such as the following:

- **Write a precise definition (in a comment) of a variable as you write its declaration.**
- **Update the definition of the variable (the comment) when you decide to change its meaning.**
- **Write a precise specification for a method (which says what it does) as you write the method's heading.**
  Do not wait until after the method is completely written or until the program is debugged.
- **Change the specification for a method whenever you make changes to the body that violate the specification.**

Preparing all the comments after the program is finished is bad for three reasons:

1. The comments were not of use to you when developing and debugging the program, so you took more time than was necessary.
2. The comments are harder to write after the program is finished, because it is difficult to remember the meaning of all the variables and methods.
3. Quite likely, you won't write the comments --even programmers with the best intentions don't spend much time filling in comments after the fact, because there are too many other interesting things to do.

You will find that writing good comments as you write a program will help you clarify your ideas and write better, correct code sooner. If you can write down clearly what your

program is doing you are more likely to have a good understanding of the problem and your program is more likely to be correct. Time spent on careful thinking and writing is more than repaid in time saved during testing and debugging.

Return to Table of Contents

Naming Conventions

Good naming conventions can help you and any reader of a program understand the program easily; bad naming conventions or no naming conventions can lead to confusion and misunderstanding. Some people advocate using very long names that define what the named entity (a method, field, local variable, etc.) represents. However, in general, this is not feasible. A good definition for an entity may require three lines of explanation or more ---how can you have such a long name? On the other hand, extremely short names --one or two characters long-- don't help at all. Thus, some in-between measures have to be adopted. There are contexts where a short name is best and contexts where a longer name is best.

Remember, a name can rarely be used to give a complete, precise definition of the entity it names, and a complete definition should always be given where the entity is defined.

The people developing Java programs have developed some conventions for identifiers, which help one readily see what the identifier represents (to some extent). You can see these conventions at work by looking at the classes in java.awt (Abstract Window Toolkit). Below, we state the conventions, with a few additions.

Package Names

Example: animalstory, java.awt

A package is a set of classes that have been grouped together. The name of a package is usually something that identifies its contents. Generally speaking, **package names should be all lower case to distinguish them from the names of the classes inside the package**. This rule holds even if the package name is a compound word.

In the Java API, you may notice that package names are separated by periods. The use of periods in package names is an advanced topic that is beyond the scope of this course; *we recommend that you avoid naming packages that way*. The period has special meaning and is not just a way of separating compound words. In particular, a word before a period is a prefix used to group multiple packages together; for example, the packages "java.awt" and "java.lang" are both grouped together under the prefix "java" (which is itself a meta-package).

Class names

Example: FilterInputStream, LivingMammals

Since a class represents a set of possible objects, each of which is an instance of the class, **a class name should generally be a noun phrase that identifies the possible objects**. A class name should begin with a capital letter, and all successive words in the class name should be capitalized.

Method names

Example: drawOval, fillOval, length, toString

A method that is **a command to do something should be given by a verb phrase that gives some indication of what the method does** (But this name should *never* be used as an excuse to not comment the method!). On the other hand, a method that implements **a function (e.g. returns a value) should be a noun phrase that describes the value**. Finally, a method that both executes some task and returns a value, you should use your best judgement in creating its name.

A method name should begin with a small letter, but all successive words in the method name should be capitalized.

(Method) Parameter Names

Examples: x, y

The precise meaning of a parameter, and any restrictions on it, should be given in the comment of the heading of the method. Therefore, particularly if the method is fairly short, **it is wise to give parameters short names**.

For a more detailed example, consider the two method headings given below. The first is preferable because it is shorter and easier to understand. Moreover, the body of the method of the first method will also be shorter and far easier to understand and manipulate.

```
// Draw an ellipse that fits exactly within the rectangle whose
// upper left corner is at position (x,y), whose width is w, and
// whose height is h. Use the current color to draw the ellipse.
void drawOval(int x, int y, int w, int h) {

// Draw an ellipse that fits exactly within the rectangle whose
// upper left corner is at position (xCoordinate,yCoordinate)
// whose width is width, and whose height is height. Use the
// current color to draw the ellipse.
void drawOval(int xCoordinate,int yCoordinate,
                    int width, int height)
```

A name like "theLoopCounter" or "firstNumber" instead of "k" or "x" causes clutter.

A parameter used as a flag should be named for what the flag represents, like *noMorePizza*, rather than simply *flag*. In addition, you should **avoid generic names like "count" and "value"**; instead, describe the items being counted or the value stored in the parameter.

Field Names

Examples: size, xCoordinate, noLines

A field name, or instance variable as they are sometimes called, contains information that helps describe the state of the object in which it occurs. Hence **a field name should be a noun phrase that describes the information the field contains** (However, the field still needs a more precise comment that describes it.) All words in a field name, *except the first*, should be capitalized.

Local Variable Names

Examples: size, xCoordinate, noLines

A local variable of a method contains information that helps describe the state of the method during its execution. **A local-variable should be a noun phrase that describes the information that it contains** (However, the local variable still needs a more precise comment that describes it). All words in a local-variable name, *except the first*, should be capitalized.

If the body of a method is short, or the places in which a local variable is used is fairly short, then a short, one-or-two letter name can be used for the local variable (see also the conventions for parameter names). A name like "theLoopCounter" or "firstNumber" instead of "kk" or "x" causes clutter. If the local variable is used only in a short context, and if it is suitably defined with a comment at its place of declaration, then use the short name.

A variable used as a flag should be named for what the flag represents, like *noMorePizza*, rather than simply *flag*. In addition, you should **avoid generic names like "count" and "value"**; instead, describe the items being counted or the value stored in the variable.

Accessor Names

Examples: getTextLength, setTextLength

An accessor method for a field either returns the value in the field or stores a value in the field. In class, we refer to these as "getters" and "setters" because of the standard Java convention. Getters are used to reference a field, while a setter is used to store the value. The names for these types of methods comes from the naming convention itself. **Getters capitalize the first letter of the field and add a "get" prefix.**; **Setters capitalize the first letter of the field and add a "Set" prefix.**;

We illustrate the naming convention with an example. Suppose that that we have a field named "textLength". To reference the field, we use the method "getTextLength()". To store a value in the field, we use the method "setTextLength()".

Constant Names

Examples: PI, Y, E, WINDOW_SIZE

In Java, a constant is simply a field of a class that has been declared final; this means that it can never change. In order to distinguish a constant from an instance variable or local variable, we **only capital letters in the names of constants**. **If a constant is a compound word, separate adjacent words in a constant name with an underscore**.

Return to Table of Contents

Indentation

Indentation is used to make the structure of a program clear. The basic rule is as follows:

> **Substatements of a statement or declaration should be indented**.

For example, the body of a method, the if-part and then-part of a conditional statement, and the body of a method should be indented.

There are several methods for placing curly braces that delimit statements; no one likes them all and all of them are disliked by somebody. Curly braces are important for both conditionals and method bodies.

First, we show off three ways to indent conditionals. For each one, we explain the pros and cons to the right.

(0)	`if (x < y)`
	`{`
	`  x = y;`
	`  y = 0;`
	`}`
	`else`
	`{`
	`  x = 0;`
	`  y = y/2;`
	`}`
		Pro:
		It is easy to match up braces
		Cons:
		Too many lines are used, and the number of lines is a scarce resource on the monitor.
		
		The substatement {x= y; y= 0;} has not been indented (the braces themselves are part of the substatement). Therefore, the basic rule concerning indentation has not been followed.

(1)	`if (x < y) {`
	`  x = y;`
	`  y = 0;`
	`} else {`
	`  x = 0;`
	`  y = y/2;`
	`}`
		Pros:
		There are no wasted lines.
		
		The opening brace is out of the way and the closing brace indicates nicely the end of a substatement.
		
		Con:
		It is difficult to match up beginning and end braces.

(2)	`if (x < y)`
	`  {x = y;`
	`  y = 0;`
	`  }`
	`else`
	`  {x = 0;`
		Pros:
		There are no wasted lines.
		
		The basic indentation rule is followed.

	`y = y/2;` `}`	It is easy to match up beginning and end braces and easy to see where a substatement ends. **Con**: The first statement of a substatement indented differently from the rest.

In this course, **we ask you to use style (1) consistently in all your programs**. This is the official recommendation by Oracle/Sun for all Java programs. However, when working on a program written by someone else, you should always match their style.

Next we show two ways to place the curly braces surrounding the body of a method; there are others.

(0)	`// Specification of method`
	`void drawLine(int x1, int y1, int x2, int y2) {`
	`  statement 1;`
	`  statement 2;`
	`}`

(1)	`// Specification of method`
	`void drawLine(int x1, int y1, int x2, int y2)`
	`  {statement 1;`
	`  statement 2;`
	`  }`

We ask you to use style (0) in this course.

Return to Table of Contents

Statement-Comments

Just as the sentences of an essay are grouped in paragraphs, so the sequence of statements of the body of a method should be grouped into logical units. Often, the clarity of the program is enhanced by preceding a logical unit by a comment that explains what it does. This comment serves as the specification for the logical unit; it should say precisely what the logical unit does.

The comment for such a logical unit is called a "statement-comment". **This comment should be written as a command to do something**. Here is an example.

```
// Truthify x >= y by swapping x and y if needed.
   if (x < y) {
       int tmp = x;
       x = y;
       y = tmp;
   }
```

The comment should explain what the group of statements does, not how it does it. Thus, it serves the same purpose as the specification of a method: it allows one to skip the reading of the statements of the logical unit and just read the comment. With suitable

statement-comments in the body of a method, one can read the method at several "levels of abstraction", which helps one scan a program quickly to find a section of current interest, much like on scans section and subsection headings in an article or book. But this purpose is served only if statement-comment are precise.

Statement comments should be complete. For example, the following comment is inadequate:

```
// Test for valid input
```

What happens if the input is valid? What if it isn't -- is an error message written or is some flag set? Without this information, one must read the statements for which this statement-comment is a specification, and the whole purpose of the statement comment is lost.

Placement of Statement-Comments

In the example above, and in the following example, the statements specified by a statement-comment are indented. In a program with several levels of statement-comments, this indentation is useful in clarifying the structure of a program. In fact, with several levels of statement-comments, even judicious use of blank lines cannot eliminate all ambiguity concerning what statement belongs to what statement-comment. The program fragment given below illustrates this:

```
// Truthify the definition of t; return false if not possible
   // Eliminate whitespace from the beginning and end of t
      while (t.length() != 0 && isWhitespace(t.charAt(0))) {
          t= t.substring(1);
      }
   // If t is empty, print an error message and return
      if (t.length() == 0) {
          ...
          return false;
      }

   if (containsCapitals(t)) {
       ...
   }

// Store the French translation of t in tFrench
   ...
```

At the highest level, this program fragment consists of two statements: (0) Truthify the definition "t" and (1) Store the French translation of "t" in "tFrench". The statement "Truthify the definition of t" is implemented in three steps, two of which are themselves statement-comments. Thus, this program fragment has three levels of abstraction.

In the example above, note how reliance on the definition of "t" in the statement-comment "Truthify the definition of t" allows the statement-comment to be short but precise. Of course, a suitable definition for "t" must appear at its declaration.

Most people prefer not to indent substatements of a statement-comment and to rely on blank lines to separate the end of the substatements of a statement-comments. Below, we show the above program fragment in this style. Note the extra comment to help the reader see the end of the substatements for "Truthify the definition of t". This style is has

its own problems. If you decide to use this style, then program in such a way that statement-comments do not appear within statement-comments, so that such awkwardnesses do not arise.

```
// Truthify the definition of t; return false if not possible
// Eliminate whitespace from the beginning and end of t
while (t.length() != 0 && isWhitespace(t.charAt(0))) {
    t= t.substring(1);
}
// If t is empty, print an error message and return
if (t.length() == 0) {
    ...
    return false;
}

if (containsCapitals(t)) {
    ...
}

// (End of Truthify the definition of t)

// Store the French translation of t in tFrench
   ...
```

Return to Table of Contents

Method Specifications

Every method should be preceded by a comment giving its specification. This specification, together with the heading of the method, which gives the number and types (classes) of the parameters and the type (class) of the result, should provide all of the information needed to use the method --and no more. It should describe any restrictions on the parameters and what the method does, not how it does it. One should never have to look at the body of a method to understand how to use it.

The specification comment should also describe the parameters of the method. The specifications can usually be written in few sentences. Here is an example:

```
/** Print on System.in the most frequently occurring temperature
    in t[0..c-1]. If there is more than one possibility, print
    the least temperature.
 */
void findCommon (Temperature [] t, int c)
```

Unfortunately, it is more typical to find someone write a comment like the following (if any comment is provided at all).

```
  // Find most frequent temperature
void findCommon (Temperature [] t, int c) {
```

This specification fails to say what part of array t is to be included in finding the most frequent temperature. It also fails to say where to print, which will be a problem if there is more than one possibility. The only way for the user to find out is to look at the body of the method (if it is available), and that should not be necessary.

This is one of the reasons why we recommend that you **use multi-line comments instead of single line comments for specifications**. It encourages you to give better detail, instead of omitting information when you reach the end of the line.

With that said, long paragraphs are just as hard to read as code. When you write comments they should be as short and to the point as possible. For a function (i.e. a method that returns a value) it is often easiest to simply describe the value returned, using the word "yield":

```
/** Yield distance between points (x1,y1) and (x2,y2) */
double dist (int x1, int y1, int x2, int y2) {
```

Another good rule of thumb comes from *The Elements of Style*, a famous little book on writing style by Cornell Professors W. Strunk, Jr., and E.B. White. Should should always use **active voice** whenever possible to keep the information short and direct. For example, do not write the specification ``This function searches list x for a value y and ...'' or ``Function isIn searches list x for a value y ...''. Such specifications are too wordy and are not commands but descriptions. Instead, they should say the following:

```
/** Yield "y is in list x" */
boolean isIn(int y, List x) {
```

Return to Table of Contents

Class Specifications

In Java, typically, each class is given in a separate file. The beginning of the file should contain a comment that explain the what the class is for. Often, one also puts here information concerning the author, the date of last modification, and so on. Here is an example:

```
/** An object of class Auto represents a car.

    Author: John Doe.
    Date of last modification: 25 August 1998
 */
public static class Car;
```

Return to Table of Contents

Variable Definitions

Every significant variable and data structure needs a precise and complete definition, which provides whatever information is needed to understand the variable The most useful information is often the *invariant properties* of the data: facts that are always true except, perhaps, momentarily, when several related variables are being updated.

> **Important Hint**: Write the definition of a set of variables when you first conceive of using them, and type the definitions as comments into the program when you type in the declarations of the variables. If you later decide to change

the meanings of the variables, change the definitions *before* you change the statement that use the variables.

Here is an example of a definition for two variables "i" and "currentItem"; note that they are defined in a single comment because they are related.

```
// 0 <= i <= currentItem < numberItems and i is the smallest value
// such that item i's price is at most item currentItem's price.
```

The definition for a boolean variable is usually best presented as the value of some English (or mathematical) statement. For example, the following two definitions are equivalent, but the first is shorter.

```
boolean b; // = " user selected menu item File | Quit"

boolean b; // true if the user selected menu item File | Quit;
           // false otherwise
```

The more precise a definition, the better. Comments like "flag for loop" or "index into b" are useless; they only say how the variable is used, but not what it means.

Related variables should be declared and described together. For example, the definition of a table should describe not only the array that holds the data but also the integer variable that contains the number of items currently in the table. In the example below, for utmost clarity tabs are used to line up identifier names and to line up the comments.

```
final int maxTemp= 150;  // max number of temperature readings.
      int nTemp= 0       // Temperature readings are in
                         // temp[0..nTemp-1], where
      double temps= new double{(maxTemp); // 0 <= nTemp < maxTemp
```

If the comment is too long to fit nicely on the right, then put it above the declarations and indent the declarations:

```
// Temperature readings are in temp[0..nTemp-1], where
// 0 <= nTemp <= maxTemp and maxTemp is the maximum number of
// temperature readings
   final int maxTemp = 150;
         int nTemp   = 0;
         double temps= new double[maxTemp]
```

Return to Table of Contents

Acknowledgments

The ideas in this missive originated in the structured-programming movement of the 1970's. They are every bit as applicable today. Many examples and ideas were taken from old Cornell CS100 and and CS211 handouts, originating in the work of Richard Conway and David Gries (see their 1973 text on *An Introduction to Programming, using PL/C*). They are also discussed in the text *Multimedia Introduction to Programming Using Java*, by Paul and David Gries. Hal Perkins has a hand in writing an earlier version this document.

Return to Table of Contents

Reading and Writing Files

This lab discusses input (e.g. reading a file). After the lab, study section 5.9 of the text. Better yet, listen to the lectures on lesson page 5-7 of the ProgramLive CD. The lectures are much clearer than the paper version.

Start this lab by downloading file lab10.zip putting its contents in a new directory. It contains these files:

- Reading.java
- cms.txt
- peop.txt

Then open the .java file in DrJava and compile it. This is the file you will be working on for this lab.

Understanding Streams

A "stream" is a sequence of data values that is processed — either read or written — from beginning to end. When the data is being read, or input, the stream is called an "input stream"; when it is being written, or output, the stream is called an "output stream". Input/output of streams is done in Java using classes in package `java.io`. **Remember to import this package when you try anything in the Interactions pane below**.

The basic way to create an input stream for a file is by creating an instance of class `FileReader`:

```
FileReader fr= new FileReader(an arg that describes what file to read);
```

And the standard way to read using `FileReader fr` is to read one character at a time, using function

```
fr.read()
```

This is too low-level for us. We would like to be able to read not one character at a time but one line at a time. For this, we use class `BufferedReader`. Instead of the above, use this:

```
FileReader fr= new FileReader(an arg that describes what file to read);
BufferedReader br= new BufferedReader(fr);
```

Then, execution of

```
String line = br.readLine();
```

reads the next line of the file and stores it in variable `lin` — if there are no more lines to read, `null` is stored in `lin`. We will see how to use this later.

Using a JFileChooser dialog box

In order to read a file, you have to indicate which file should be read. The easiest way to do this is to use a dialog window to navigate to the appropriate directory and select the file. For this, we use an instance of class `JFileChooser`, in package `javax.swing`. Execute the following in the Interactions pane:

```
br= Reading.getReader(null);
```

A dialog box opens. Its title is "Choose input file". And it allows you to navigate anywhere you want and then select a file. Do a bit of navigating and select a file. Then take a look at function `getReader(String)`. Here is what it does:

1. Declare local variable `jd` of class `JFileChooser`.
2. Store in `jd` a new instance of class `JFileChooser`. In the new expression `new JFileChooser()`, you have no control over the directory that appears in the dialog window initially. In the new e xpression `new JFileChooser(p)`, String `p` is supposed to be a path on your computer of the directory to open in the dialog window. This choice allows you to dispense with a lot of navigating.
3. Set the title of `jd` to "Choose input file".
4. Execution of `jd.showOpenDialog(null);` causes the dialog window to open on your monitor, and the program pauses until you have closed it. Nothing happens until you have chosen a file (or canceled the interaction).
5. Create a new `FileReader` with the file that you selected (its name is `jd.getSelectedFile()`) as the argument and store its name in `fr`.
6. Create and return a `BufferedReader` that is attached to `fr`.

The function for obtaining the next line from a `BufferedReader br` is:

```
br.readLine()      // = the next line of BufferedReader br —null if there are no more lines
```

In the Interactions pane, you can continue to evaluate `br.readLine()`. Each time you do it, the next line of the file you selected is printed. Try it.

Processing the lines of a file

Function `lines` in class `Lab10` illustrates the basic way of processing the lines of a file given by a `BufferedReader`. In the Interactions pane, put a call on this method and let it read some file —you will see how many lines the file has in it.

Study this method. Any loop that you write that processes a file should be similar to this one —the "processing" of each line will change, but the basic structure of the loop that does the processing will not. Note that this method contains a while-loop. Here are important points:

1. The first line is read before the while loop and stored in variable `lin`. `lin` will be `null` if the file is empty.
2. The loop stops when `lin` is `null`, indicating that there are no more lines in the file.
3. The repetend first processes the line given by variable `lin` and then reads the next line into `lin`.

The header of the method contains a new construct: `throws IOException`. It is needed because function `br.readLine()` might create some sort of I/O (input/output) error, and this is how we handle it. We talked about exceptions previously in the course.

Write Your Own Method

Reading a File

Write the body of function `contains` that is declared in class `Reading` (and test it). If you are smart, you will write a JUnit testing class with a procedure to test this method, though that is not required.

The specification of this method is including in `Reading.java`. The method takes a parameter `s` and checks to see if the value in this parameter is a line in the file of the user. For example, we could use the method call

```
Reading.contains("djg17")
```

to see whether the Cornell netid `djg17` appears in one of the two files `cms.txt` and `peop.txt`.

This is the only part of the lab that is required. Please show your code to you instructor when you are done.

Writing files

This section describes that we had to do in previous semesters when dealing with prelim conflicts. The task is to read in two files of netids, say `cms.txt` and `peop.txt` (containing netids for our course and netids for a particular math course), and create a file that contains all netids that are in both files. You can assume that the two files are already sorted by netids. This is nice example works with input/output files and also contains an interesting loop. It can be simple or comlpicated, depending how **you** write it.

Please read on and learn about writing files, but you do not have to complete this task for this lab.

Writing files is not much different from reading them. Procedure `getWriter` obtains the name of a file (in a particular directory) to write, using a `JFileChooser` window that is placed on the monitor using `showSaveDialog` (instead of `showOpenDialog`). The procedure uses the resulting file as an argument in the constructor call of a `PrintStream` object.

Try this in the Interactions pane:

```
PrintStream v= Reading.getWriter();
```

When the dialog window opens, navigate to a directory and select a file name or type in a completely different one; then you can select one and then change it slightly. **Be careful**. You will not want to overwrite an existing file, but this can easily happen.

Now, type

```
v.println("first line in the file");
v.close();
```

The first statement writes a line to the file. You can write as many lines as you wish. The second statement should be used to close the file.

After these statements are executed, look at the directory where the file was written, open the file, and look at it.

Now, write (and test) the body of procedure `Reading.extract`. Note that it has a series of comment-statements; each is a statement to do something, written in English. Write the Java implementation of each comment directly under the comment and follow the implementation with a blank line.