COMP 111 Reading Guide:
Big Java, 3rd Edition

This reading guide is intended to serve several purposes:

- First, it provides a summary of the readings in each section of the textbook using different language and examples from the author. It is most certainly not, however, a substitute for reading the text because few, if any, coding examples or in-depth explanations are provided.

- Second, it indicates which sections of the text are particularly important. In reality, very few sections are unimportant, but some require more careful study.

- Third and finally, it serves as a quick reference for exam preparation. However, when using it for that purpose, please be aware that it is much easier to *recognize* the correct answers than it is to *produce* them. One caveat is that this guide only covers the main presentational sections in the textbook. There are many Advanced Topic, Common Error, Productivity Hint and Random Fact sidebars that contain important and useful information, and these sections should also be read and studied.

Chapter 1: Introduction.

Section 1.1: What is Programming? *Programming* is one activity within the larger context of Computer Science. A computer program is a sequence of simple instructions that combined can carry out complex tasks very rapidly. A programmer takes the complex tasks and breaks them down into simpler tasks and writes the instructions that the computer carries out.

Section 1.2: The Anatomy of a Computer. The simple instructions of the program are executed on the *central processing unit* (CPU). The programs and data are stored on disk and loaded into memory in order for the CPU to process them. Instructions within each program direct the CPU to read from memory, write to disk, display images on the screen, or communicate over the network.

Section 1.3: Translating Human-Readable Programs to Machine Code. Even though the CPU only understands simple instructions such as addition, comparisons and reading or writing to memory, programs are typically written at a much higher level. Programmers use a language such as Java to describe the tasks, and a compiler translates the high level language into machine code. Java uses a *virtual machine* (the JVM) to interpret the compiled code on a real CPU.

Section 1.4: The Java Programming Language. The JVM is itself a program written in C and C++ that runs on many different machines and operating systems. Once a JVM is available on a new computing device, then a Java program written on any other machine will run on the new device. After Sun introduced Java in 1995, Java applets quickly spread as extensions to browsers. However, Java's real potential

wasn't tapped until it became part of server environments. As an "industrial" language, Java poses two problems for students: it is not simple to write basic programs, and the vast libraries are intimidating to approach. The goal of this course is to teach object-oriented principles, not Java specifically.

Section 1.5: Becoming Familiar with Your Computer. *Integrated development environments* (IDEs) are software programs used to design, implement, test, and debug other programs. Many different industrial-strength environments exist for the Java language, including NetBeans, JBuilder, IntelliJ and Eclipse. For the purposes of this course, we will be using BlueJ – a simplified environment designed specifically for teaching object-oriented principles in Java. In any computing environment, having backups of the important work is a good idea as an unexpected event can cause the loss of hours, if not weeks, of effort.

Section 1.6: Compiling a Simple Program. As mentioned in the previous section, Java has a significant amount of overhead to get even a simple program to work. Certain words are reserved by the language, and all of the language is sensitive to the capitalization of words. Writing a program so that it is properly formatted with indentation and placement of curly braces helps to be able to read it later. In Java, each program is contained in one or more *classes*. Within this class, we create *methods*, which define behaviors within a class. The `main` method is the point at which the program begins execution. Methods can create objects and invoke other methods defined in those classes. For instance, within the `System` class there is an `out` object, which contains a `println` method. The `println` method takes a single parameter of the string to print to the screen.

Section 1.7: Errors. When writing any code, there are two different kinds of errors a programmer can encounter: *syntax errors* and *logic errors*. Syntax errors are equivalent to grammatical errors in English – the spelling or order of the words is incorrect. The compiler will identify lines on which syntax errors occur so that the errors can be corrected. Once the program is grammatically correct, there can still be logic errors. Programming logic errors are like logic errors in English – the sentence or paragraph simply does not make sense or convey the message that the author intends. Likewise, it is possible to write programs that are grammatically correct that do not do what the programmer wanted but crash or create incorrect output. Logic errors can only be found by running the programs and testing their results and, therefore, are much more difficult to find.

Section 1.8: The Compilation Process. Writing and testing programs in any language requires many steps. Though integrated development environments simplify the process, it is essentially unchanged from 30 years ago: use an *editor* to create a text file containing program source code; use a *compiler* to detect syntax errors and translate the source code into binary (machine) executable code; run the resulting executable to detect any errors; repeat the cycle to correct any syntax or logic errors found.

Chapter 2: Using Objects.

Section 2.1: Types and Variables. Java is both a strongly and statically typed language, meaning that all variables in the language have a *type*, and that type is declared when the variable is created. For instance, a variable created to hold an integer value can only ever hold an integer and never a string or floating-point value. A variable has more properties than just type; it also has properties of its name (an identifier), its value (the contents it holds in memory), and its scope (the range of lines in the code where it is visible).

Section 2.2: The Assignment Operator. The *assignment* operator = is used to associate a variable with a value stored in memory. Before any variable can be used (i.e. printed to the screen, used as a value in a formula, etc.), it must be *initialized* with a value through assignment.

Section 2.3: Objects, Classes, and Methods. *Classes* are templates for creating objects just as a blueprint is a template for creating a building. A class defines which *instance fields* (attributes) and *methods* (behaviors) will be available for each object that is built from its template. An *object* is an instance of a class just as a particular building is an instance of a blueprint. Many buildings can be created from a single blueprint and likewise many objects can be built from a single class. However, each building is unique in its *identity* and each object is likewise unique. Just as each building created from a particular blueprint has the same general form but varies in its colors, trim, and fixtures, each object may have different *values* for its attributes. In a like manner, methods define behaviors of or actions taken on an object, and are called the *interface* to the object.

Section 2.4: Method Parameters and Return Values. Similar to the imperative form of English sentences, methods take on the form of "*noun.verb(extra_information)*". Commanding a dog to fetch a stick in English would be written: "Spot, fetch the stick." In Java this would be written `spot.fetch("stick");` and `spot` is an object of class `Dog`. The extra pieces of information needed by a method to carry out its actions are the *parameters* to the method. Parameters can be any variable, expression, constant, or object. Methods may have a *return value* as the result of the action. Thus, if the fetch method of the `Dog` class returns a Boolean value indicating whether `spot` obeyed, we can use that information in further processing.

Section 2.5: Number Types. Number types in Java fall into two separate categories: *integer* types and *floating-point* types. The two most common types are `int` and `double`, respectively. Floating point types can be used to express fractional parts, such as `3.1415`, whereas integer types correspond to the whole numbers such, as `42`. Number types are distinct from classes; they are not manipulated by methods called on them but rather by operators that combine them. Though number types cannot be used as the target of method calls, they are frequently used as parameters to methods and as instance fields within classes.

Section 2.6: Constructing Objects. Objects are *abstractions* of real-world ideas. For instance, an object of class `Rectangle` represents or describes the concept of a rectangle but is not itself a rectangular shape. Objects are described by their instance fields and methods. For instance, a `Rectangle` can be described by a

point (`x`, `y`) and a `width` and a `height`. However, those are the internal details of the object that are typically hidden from programmers using the `Rectangle` class, and the original `Rectangle` programmer could have used two points (`x1`, `y1`) and (`x2`, `y2`) to describe the object. To create a `Rectangle` object, the programmer must use the operator `new` followed by the class name to call the *constructor* of the class. Constructors are special methods that are responsible for initializing objects and typically use the parameters passed into them to perform the initialization.

Section 2.7: Accessor and Mutator Methods. As stated previously, objects are manipulated through the interface provided by the defined methods. A method that does not change the state of an object (its instance fields) is called an *accessor method*, and a method that does change the data within an object is called a *mutator method*. Frequently programmers follow a naming convention defined by the JavaBeans standard in which accessor methods start with the prefix `get` and mutators start with `set`. For instance, if a `Person` object had an instance field `name`, then the `getName()` method would extract data from the object without changing its state, and the `setName()` method would alter the state of the object by replacing the existing name with a new one specified as a parameter. Not all accessors and mutators follow this convention, however, and it is important to consult the documentation for the class in order to tell the difference.

Section 2.8: Implementing a Test Program. One of the most important tasks facing a programmer is assuring himself and others that the code is correct. One way to do that is to write *unit tests*. Unit tests are themselves programs whose job it is to call other methods and ensure that the actual output matches the predicted output. The textbook takes the approach of manual validation by printing out the results of the method calls and relying on the programmer to observe when they are incorrect. However, a more sophisticated method is to have unit tests perform the comparison internally and only print something to the screen when the test fails. This is the approach that *JUnit* takes. JUnit is a testing framework into which you place methods that will test other methods. (See the Appendix for specifics about JUnit.) Java classes are grouped into packages, and so to test classes from within the JUnit framework, it is necessary to use the keyword `import` followed by the class name you wish to use to allow one class access to another.

Section 2.9: The API Documentation. In learning any new programming language, there are two basic tasks: mastering the language itself and mastering the pre-written *libraries* provided by the language authors. The Java libraries are vast and actually take more time to learn than the constructs of the language. The *application programming interface* (API) of the Java libraries is described in an extensive collection of HTML documents available for browsing over the Internet or downloaded onto your computer. In the API, each class is documented in terms of its intended use, the methods provided, and the parameters to each method. Many entries also contain warnings about pitfalls and example code to show how the methods should be used. Browse to the Java API Documentation and drill down to locate classes of interest.

Section 2.10: Object References. Unlike with *primitive types* (i.e. built-in types) such as numbers and characters, a variable whose type is a class does not actually hold the data of the object. Instead, the variable holds the location of the object that exists somewhere else. Java, like many other programming languages, calls this a *reference* to an object. A reference is similar to a street address for a building – the address uniquely identifies the location of a real world physical object. In the same way, a reference identifies the location of location of a software abstraction of an object. Just as two or more letters can be addressed to the same building, two or more references can refer to the same object. Assignment of one reference over top of another reference merely copies the *location* of the object from one to another, and thus both refer to the same object. With primitive types, however, the actual data is copied, and so if one primitive is assigned over top of another, then two copies of the data exist in the program.

Chapter 3: Implementing Classes.

Section 3.1: Levels of Abstraction. A *black box* is any device whose inner workings are hidden. To many people a computer is a black box because they have no knowledge of how or why it works the way it does. They know, however, that they can type at the keyboard and see results on the monitor, and they can install new programs by inserting a CD-ROM. To an IT specialist, the computer is less of a mystery: it contains a hard disk, CD recorder, video card, motherboard, RAM, and processor, among other things. To that same IT specialist, though, the processor is itself a black box – but not to an electrical engineer. A black box provides *encapsulation*, hiding unimportant details from people who don't need them. It allows a computer user to interact with the computer at one level and a computer repair person to interact with it at another level. In order to achieve this, though, somebody needed to *abstract* out the essential concepts and provide a usable *interface* to the various users. Consider the API documentation from Section 2.9 – the documentation is written to a programmer who uses the component as a black box in an application. The component itself, however, was written by another programmer. Thus, the job of a programmer is two fold: write software that uses other programmers' components and write components that others can reuse in their software. Object orientation is the main *implementation* method today for both of these tasks.

Section 3.2: Specifying the Public Interface of a Class. The process of *abstraction* allows a programmer to find the essential feature set of an object. Essential features are the methods (behaviors) of the object that will be needed to solve the problem the programmer is facing. Some behaviors will be important for certain kinds of problems and other will be irrelevant. For instance, given a `Car` object, an `accelerate()` behavior would be important for a racing game but would be irrelevant for a vehicle registration system. Methods consist of five separate pieces of code: the access specifier, the return type, the method name, the parameter list, and the method body. The *access specifier* determines which other classes can invoke the method: `public` means that all classes may invoke it, whereas `private` means that only other methods in the same class can invoke it. The *return type* determines what kind of data is to be returned, such as a numeric type, another object, or `void`

if nothing is returned. The *method name* should be descriptive of the behavior, and is usually a verb or verb phrase. The *parameter list* provides zero or more pieces of additional data needed to carry out the behavior. Finally the *body* of the method is the actual program code that implements the behavior. *Constructors* are similar to methods in that they have an access specifier, parameter list and body. However, constructors are different in that their job is not to define a behavior but rather to properly initialize an object for later use. Constructors are always named the same as the class, and constructors never have a return value. A class is a construct that holds all the methods, constructors, and instance fields. Classes, too, have access specifiers, but generally are `public`.

Section 3.3: Commenting the Public Interface. Since part of a programmer's job is to create objects that other programmers can use and reuse, the task of documenting how to use the objects becomes extremely important. Most modern languages provide a mechanism where specially formatted *comments* inserted into the program code can be extracted into API documentation. Comments are notes written by the programmer to other programmers (self included) that describe aspects of the code. API documentation describes the purpose and use of the class and its methods. In Java, such JavaDoc comments fall between the `/**` beginning marker and the `*/` ending marker. Special tags in the comments denote particular fields of interest. For example, `@param` describes a parameter and `@return` describes the return value. Comments that are not destined for API documentation describe how or why certain decisions were made. These comments are intended for someone maintaining the class rather than someone who is using the class. These comments begin with `//` and last until the end of the line.

Section 3.4: Instance Fields. Instance fields are pieces of data needed to implement the behaviors of the methods of a class. Each object that is created from the class blueprint will have a copy of these variables – they aren't shared between objects. However, consider that the principle of encapsulation is concerned only with *what* the object does, not with *how* it does it. To expose the way in which something is implemented would be to turn the black box into a clear box. Hiding the details of how things work is an important concept in object-oriented programming. Thus, unlike local variable definitions, which have only a type and name, instance fields also have an access specifier. Typically, this specifier should be `private`, not `public`. When a `private` access specifier is used, no code *outside* this class can access the element tagged as such. However, within the class, all the methods and constructors can read and write the `private` field. A final note about encapsulation is that it is far easier to take a `private` method or field and make it `public` at a later time than it is to take a `public` method or field and make it `private`. Anything that is `public` can (and likely will) be used by other programmers reusing a class, so suddenly changing something from `public` to `private` will cause their code to no longer compile.

Section 3.5: Implementing Constructors and Methods. The last piece of writing constructors and methods is the body of code they contain. The code within each is an implementation of an *algorithm* – a step-by-step procedure for solving a problem. This piece is likely the most challenging part of software development. An

algorithm is similar in some respects to a recipe for cooking in that it defines a sequence. However, unlike a recipe, many algorithms involve conditional and repetitive execution (covered in Chapters 6 and 7 respectively). At this point in the course, however, the method bodies will consist of mostly sequential statements involving arithmetic and assignment operators. Finally, if a method should produce an answer, a `return` statement followed by a value that matches the return type is used, and that result will be handed back to the caller of the method.

Section 3.6: Unit Testing. As mentioned in Section 2.8, testing assures the logical quality of the code a programmer writes. In BlueJ, it is possible to interactively test classes by creating an object and then "recording" interactions with it. The return values of method calls can be compared against the predicted values. When the values don't match, then the JUnit testing framework will show a red bar. When all the return values match what is predicted, then the framework will show a green bar.

Section 3.7: Categories of Variables. Thus far, three separate categories of variables have been discussed: instance fields, local variables, and parameter variables. (A fourth type, static fields, will be discussed in Section 9.7.) All variables have several properties: name, type, value, scope, and lifetime. Of these, the first three have already been covered, but scope and lifetime are the distinguishing factors between instance fields, local variables, and parameter variables. *Scope* is the range of lines in program code where the variables are visible and available to be used in expressions. *Lifetime* is the span of time while the program is executing when the variables exist in memory. The two concepts are linked, yet distinct. Scope is a concern at the time that you are writing the program, and lifetime is a concern as the program is running. The lifetime of objects is determined by how long they are in active use in the program. If a parent object holds a reference to a child object in an instance field, then as long as the parent is alive, the child will be as well. Since there can be multiple references to a single object, as long as even one is active, then so is the referenced object. However, when there are no longer any active references to an object, then the Java *garbage collector* is free to reclaim the memory that was occupied by the dead object.

Section 3.8: Implicit and Explicit Method Parameters. When calling a method on an object by the `noun.verb(extra_information)` metaphor, the *explicit parameters* to the method are the extra information required to carry out the behavior. However, there is yet another parameter that is passed – the *implicit parameter*. The implicit parameter to every method is the object on which the method is invoked (the object on the left-hand side of the dot separating noun from verb). Sometimes it is useful to be able to access the implicit parameter explicitly. Within each method, the object on which the method is invoked is known by the keyword **`this`**. For example, consider the method call `spot.fetch("stick")`. From within the `fetch` method, **`this`** is a reference to the `spot` object at runtime. Many programmers get in the habit of using the **`this`** reference when accessing other methods or instance fields as it adds clarity to their coding style. One particularly important use of **`this`** is when the programmer wants to use one constructor to do the initialization for another constructor in the same class. For

example, given a constructor for class `Dog` that has a signature of `public Dog(String breed)`, and you wish to also have a zero-argument constructor **`public`** `Dog()` that defaults the breed to Beagles, then inside the zero-argument constructor you may have the single line: `this("Beagle");` to call the one-argument constructor that actually does the initialization.

Chapter 4: Fundamental Data Types.

Section 4.1: Number Types. Number types fall into two categories: integer types and floating-point types. Integer types represent countable quantities and are available in various number ranges depending on the size of the variable. From smallest to largest these types are **`byte`**, **`short`**, **`int`**, and **`long`**. The floating-point number types allow representation of fractional quantities and also have a range depending on size. The available types are **`float`** and **`double`**. By far, **`int`** and **`double`** are the most frequently used. Be aware that there are difficulties in using number types – overflow and rounding errors are common. Overflow occurs when the result of the computation won't fit back into the same number type, as might occur when multiplying two **`int`** types together. Rounding errors occurs because, in the binary number system of computers, decimal fractions frequently don't have a finite non-repeating representation. Also, while it may seem that choosing a number type is easy, there are pitfalls. For example, representing money seems to call for a double, but in reality, the decimal point in money doesn't float – it's fixed and, therefore, money would best be represented by an integer. "Smaller" number types can fit into "larger" number types and be implicitly converted. Thus, it is always possible to convert a **`short`** to an **`int`**. However, going the other direction may cause an overflow and, therefore, requires a *cast*. Casting is an *explicit conversion*, whereas the previous example is an *implicit conversion*. Non-number types include **`char`**, which represents character data (including international characters), and **`boolean`**, which can only hold the two values **`true`** and **`false`**.

Section 4.2: Constants. Though the term "constant variable" may seem to be an oxymoron, it is nonetheless possible to define and initialize a variable whose value cannot be changed after initialization. Prefacing the declaration of a variable with the keyword **`final`** does two things: first it forces the variable to be initialized at the point of declaration, and second it prohibits any alterations to this variable in later program code. Typically, named *constants* like this are used to make program code easier to read. For instance, the `Math` library has a named constant `PI` that can be used in periodic calculations. To expose the constant, not only is it **`final`** but it is also **`public`** and **`static`**. The **`public`** property allows other classes (besides `Math`) to access the constant, and **`static`** means that the variable belongs to the class rather than to an instance of the class (an object). Thus, static variables are a kind of "global" data since every object of the class shares them rather than having a separate copy in each object as would be the case with instance fields. Named constants, by convention, are written in uppercase characters using underscores to connect words. It is important to note that a variable of class type (an object) can be made **`final`**, meaning that it will only refer to a single instance. However, if the object itself defines mutators, then the object can still be altered. Hence, only the reference is constant, not the object itself.

Section 4.3: Assignment, Increment, and Decrement. Assignment uses the operator = to overwrite a variable in memory with a different value. Assignment is a different operation than the equality comparison operation (covered in Chapter 6), and it can look somewhat confusing mathematically to read a statement like `i = i + 1`. Rather than being viewed as an equation, this statement is interpreted as "take the current value of `i`, add `1` to it, and make that the new value of `i`." This increment operation is so common that special abbreviations of it (the increment operators) are available as syntax in Java. Increment operators have two forms: pre- and post-increment. These operators have both a value and a side effect of evaluation. The side effect of both is the same: the variable is incremented by 1. However, the difference is in the value of the expression itself. Suppose that `i` has the value `3`. Then both `++i` (pre-increment) and `i++` (post-increment) would change `i` to `4`. However, the expression `++i` evaluates to `4` whereas the expression `i++` evaluates to `3`. The pre-increment operator adopts the *new* value of `i`, whereas the post-increment operator retains the *old* value of `i`. Just as there are increment operators, there are also decrement operators. For example, `--i` and `i--` are legitimate expressions indicating decrementation. An example in the table below should help clarify. Assume that, for each of these independent expressions, the variable `i` is an `int` with the value `3` at the outset.

Expression	Final value of `x`	Final value of `i`
`x = i++ + 5;`	8	4
`x = ++i + 5;`	9	4
`x = i-- + 5;`	8	2
`x = --i + 5;`	7	2

When an expression makes multiple changes to a single variable by increment or decrement, the side-effects are undefined. As a result, an expression like `(++i - i++)` has unpredictable results.

Section 4.4: Arithmetic Operations and Mathematical Functions. Addition, subtraction, multiplication, and division are all defined by built-in mathematical operators `+`, `-`, `*`, and `/` respectively. Expressions follow the order of operations common to algebra in that multiplication and division have higher precedence than addition and subtraction. To change the order of operations, group sub-expressions in parentheses. Note, however, than when dividing integers using the `/` operator, the fractional result is discarded leaving an integer result. For example `7/4` is evaluated to `1`. To include the fractional part of the quotient in the result, either *cast* one of the operands as a **`double`**, or, in the case of constants, use `4.0` rather than `4`. The modulus operator `%` yields the *remainder* of integer division. For example `7 % 4` (read as "seven mod 4") is `3`, because `7/4` is `1` *remainder* `3`. The last mathematical operator to know is the unary minus (*unary* operators only have one operand as opposed to *binary* operators, which have two). Unary minus is also known as negation, and would be written as `-x`, producing the negation of `x`. Other common mathematical operations are programmed as method calls in the `Math` class. `Math.sqrt(x)` for instance would calculate the square root of `x`. See the documentation for the `Math` class for other methods.

Section 4.5: Calling Static Methods. It may have seemed strange to call methods in the `Math` class by using a class name on the left hand side of the dot operator. Normally the left hand side of the dot contains an object name, yet no object of class `Math` was constructed. The methods defined in the `Math` class are *static methods*. Unlike instance methods, static methods are not called on an object and, therefore, have no implicit parameter **`this`**. Therefore, static methods must receive all their input from explicit parameters. The Java documentation of library classes identifies which methods are **`static`** and which are not.

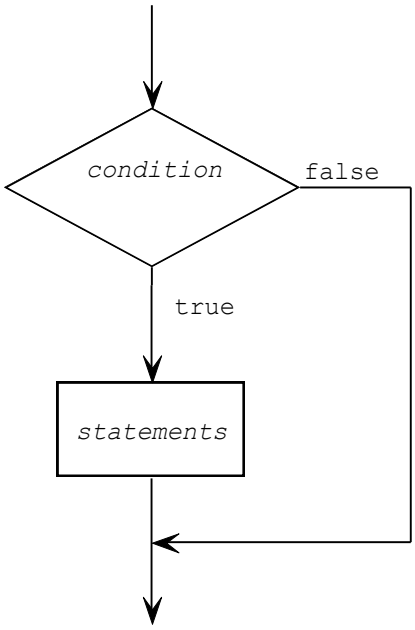
Section 4.6: Strings. A `String` object is an abstraction of text data and can easily be thought of as a sequence of characters. As an object, a `String` has both attributes and behaviors that are related to manipulating text data. However, since `String` is arguably the most commonly used data type, Java supports special syntax for the construction and concatenation of strings. A string can be constructed by simply putting some text in quotation marks. In addition, strings can be joined to other strings, numbers, and even other objects using the + operator. To do so, merely place a `String` object on the left hand side of the + operator and any other kind of data on the right hand side; the resulting object will be a joined `String` object. For example, `"COMP" + 111` would result in the string `"COMP111"`. `String` variables and `String` constants have methods defined for them, but each method is an accessor only. The `String` class defines no mutator methods and, therefore, `String` objects are called *immutable*. For example, `"HelloWorld".length()` would return the number `9` because there are nine characters in the string. For any `String` object, many methods exist, such as `substring()`, `charAt()`, `length()`, `equals()`, `equalsIgnoreCase()`, `contains()`, `indexOf()`, and many others. Consult the Java API documentation to explore these commonly used methods.

Section 4.7: Reading Input. Java was designed from the beginning to use GUI interfaces or Web interfaces for its input. As a result, reading input from the keyboard directly into a console-style program has been difficult until Java 5. However, using the `Scanner` class, it is now possible to easily read string and number types from the keyboard directly. To do so, create a `Scanner` object using the constructor, and pass in `System.in` as the parameter: `Scanner in = new Scanner(System.in);` Then, use the methods `nextInt()`, `nextDouble()`, `nextLine()`, or `next()` to read in integers, doubles, entire lines as strings, or the next word as a string, respectively. The `Scanner` class uses the `System.in` object as a source of bytes and then re-interprets those bytes as numbers or strings, depending on the method called. In the language of *design patterns*, the `Scanner` class is called an *Adapter* because it changes (adapts) one object interface into another more usable or friendlier object interface.

Chapter 5: Decisions.

Section 5.1: The `if` Statement. Programs do not execute in straight lines – they need to make decisions based on input and choose from a variety of responses decided on by the programmer. For example, a `Car` object can only `drive()` if there is gas in the tank. The `if` statement allows programs to select from a variety of alternatives based on the conditions the program checks. The syntax for an `if` statement is:

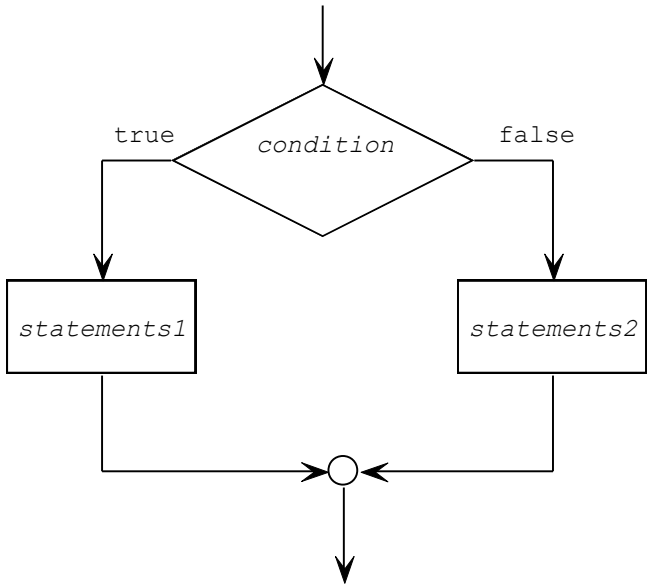
```
if  (condition)
{
    (statements)
}
```



The *condition* is an expression that returns a **`true`** or **`false`** (Boolean) value. Typically, the condition will use the *relational operators* for numbers (covered in Section 6.2) or *predicate methods* for `String` objects. The *statements* are the body of the **`if`** and define any actions that should be executed when the *condition* is **`true`**. The curly braces `{` and `}` are used to group all the statements into a *compound* or *block statement*. If only one statement is to follow the **`if`** condition, then no braces are required; however, it is a good practice to always include the braces should more statements later be inserted in the body of the `if` statement. Execution of the **`if`** statement is illustrated in the above flowchart. Note that execution splits into two alternatives at the condition and then joins again after the body of the **`if`**.

When there is more than one alternative, then the **`if`** statement can be combined with an **`else`** clause to create two mutually exclusive options. The syntax for the **`if`**/**`else`** construct is:

```
if  (condition)
{
    statements1
}
else
{
    statements2
}
```



When the *condition* is true, then *statements1* will be executed. However, when the *condition* is false, then *statements2* will be executed. Again, there are two paths through this code, and either *statements1* will be executed, or *statements2* will be executed, but not both. The flow of control will be restored to one path after one of the block statements is executed.

Section 5.2: Comparing Values. Relational operators compare two numerical values to determine a relationship between them. The operands are numerical values, and the result is a Boolean value (**`true`** or **`false`**). Because the operators produce Boolean results, such comparisons can be used as the conditions in **`if`** statements. The operators >, <, >=, <=, `!=`, and == permit comparison on greater than, less than, greater than or equal to, less than or equal to, not equal to, and equal to, respectively.

When comparing floating-point numbers, it is important to consider the ramifications of floating-point roundoff errors. Since it is possible (even probable) for two numbers to be very close without being equal, instead of testing for equal, test to see if the numbers are "close enough" to be considered equal for the purposes of the given situation.

So far, comparisons have been limited to numerical values. Strings, however, are also usefully compared. To compare two strings for equality, do not use the == or `!=` operators. Instead, use the `equals()` method. When comparing strings (or any other object) using ==, all that are compared are the *references*. Two object references are only equal if they refer to the *same* object, not if they simply contain the same data. To compare strings in a case-insensitive manner, use `equalsIgnoreCase()`. Finally, to compare strings on less than, less than or equal to, greater than, and greater than or equal to, use the `compareTo()` method: `s1.compareTo(s2)` compares strings alphabetically, and returns an **`int`** that is

either less than 0, equal to 0, or greater than 0 depending on whether `s1` is less than `s2`, `s1` is equal to `s2`, or `s1` is greater than `s2` respectively.

Objects can also be compared, but the Java language has no knowledge of what objects will be written. Additionally, it is frequently difficult to define a comparison between objects. For example, what does it mean for one `Dog` object to be "less than" another `Dog` object? Nonetheless, to compare two `Dog` objects, the `Dog` class can define its own `equals()` method and its own `compareTo()` method that follow the same contract as `String`.

An object reference that refers to no object is said to have a `null` value. Since attempting to call a method on a `null` reference will cause a program to crash, it is important to test thoroughly to avoid `null` references.

Section 5.3: Multiple Alternatives. Determining the order of conditions for correct execution is difficult since, obviously, not every arrangement will result in the right answer. A sequence of **`if`**/**`else`** decisions is sometimes necessary to choose among many mutually exclusive alternatives. For example, the following table is a schedule of fines for speeding violations:

1 – 15 MPH	16 – 29 MPH	30+ MPH
$95.00	$115.00	$155.00

The order of the **`if`**/**`else`** statements is critical in determining the correct fine. For example:

```
if (speed >= 1)
    fine = 95;
else if (speed >= 16)
    fine = 115;
else // 30+ miles over
    fine = 155;
```

Note: This code is wrong!

A speeder going 45 miles per hour over the speed limit would be pleasantly surprised to only have a $95.00 fine. The solution is to either reverse the ordering or reverse the direction of the comparison.

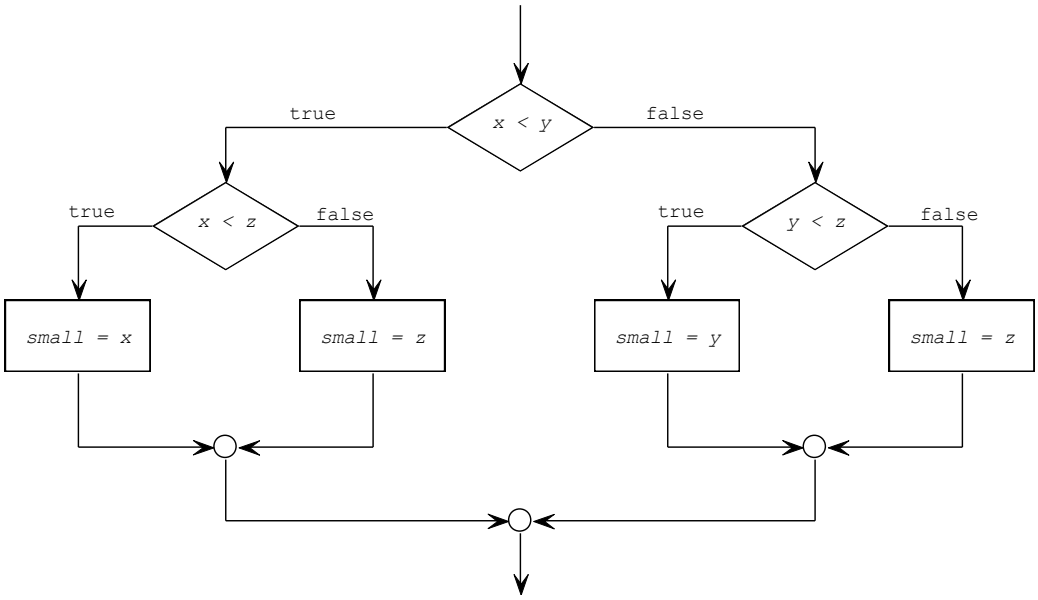
```
if  (speed >= 30
      fine = 155;
else if (speed >= 16
      fine = 115;
else // 1 - 15 over
      fine = 95;
```

OR

```
if  (speed <= 15
      fine = 95;
else if (speed <= 29
      fine = 115;
else // 30 + miles over
      fine = 95;
```

Selecting from a simple list of mutually exclusive options is straightforward. A more complicated problem involves creating *nested branches*. These kinds of

problems involve more sophisticated logic and can be both difficult to program and to read. Consider a method that is to determine the smallest of three integers *x*, *y* and *z*. A flow chart for the logic is shown below. Does the logic work when two or more of the numbers are equal?



The code for this method is easily expressed based on the logic above. The difficult part of programming isn't writing the code, but rather it is determining the logic that solves the problem.

```
public static int smallestOfThree(int x, int y, int z)
{
    int small;
    if (x < y)
        if (x < z)
            small = x;
        else
            small = z;
    else
        if (y < z)
            small = y;
        else
            small = z;
    return small;
}
```

When constructing complex logic it is crucial to know which **`if`** the subsequent **`else`** matches. The compiler ignores indentation, so formatting your code in an effort to pair an **`if`** with an **`else`** may be deceptive to the reader. An **`else`** clause *always* matches the nearest **`if`**. In order to force it to match a more distant **`if`** statement, use the compound statement to create a block (enclosed in braces). Failing to do so is called the *dangling else problem*.

Section 5.4: Using Boolean Expressions. As mentioned in Section 4.1, Java has a **`boolean`** data type. Boolean variables can only hold the value **`true`** or **`false`**. Further, if a method has a return type of **`boolean`**, it is referred to as a *predicate method*. Frequently these methods are named to begin with a question verb such as *is* or *has*. Using just **`if`**/**`else`** statements and the sequence of statements, it is possible to build very complex conditions, but the logic behind them can be difficult to ascertain. Therefore, Java defines three additional operators that combine Boolean expressions into more complex, yet easy to read expressions. These operators are `&&`, `||` and `!`; they represent logical conjunction (and), disjunction (or), and negation (not). The operations are defined by the *truth tables* below:

X	Y	X && Y	X || Y	!X	!Y
false	false	false	false	true	true
false	True	false	true	true	false
true	false	false	true	false	true
true	True	true	true	false	false

Notice that `X && Y` is only **`true`** when both `X` and `Y` are **`true`**, and that `X || Y` is only **`false`** when both `X` and `Y` are **`false`**. As with arithmetic operators, Boolean operators follow an order of operations in complex expressions: `&&` is always evaluated before `||` just as multiplication is evaluated before addition. Negation always has the highest precedence. Finally, the order of operations can be changed using parentheses.

Section 5.5: Test Coverage. Test cases are subdivided into two general categories: black-box and white-box. *Black-box tests* are written without knowledge of the internal workings of the system under test. *White-box tests* exploit knowledge of the underlying implementation. As a general rule, *unit tests* written by the programmers are white-box tests. However, *acceptance tests*, written by the "customer" or end-user, are generally black-box tests. JUnit supports white-box unit testing, whereas FitNess (an acceptance testing framework) is for black-box acceptance testing. To quote the FitNess documentation, JUnit ensures that you "build the code right" whereas FitNess ensures that you "build the right code" to solve the problem the customer wants.

Finally, it is important to know the *test coverage* of the test suite. Test coverage is a measure of how much of the application's code is actually tested by the test cases. There are many ways to measure test coverage: statement coverage ensures that each executable statement is executed at least once by tests, decision coverage ensures that each conditional expression is evaluated to both true and false by the tests, condition coverage ensures that each sub-expression in a Boolean condition evaluates to both true and false, and, finally, path coverage ensures that each logical path through a function is exercised. It should always be a goal to get as close to 100% statement coverage as possible.

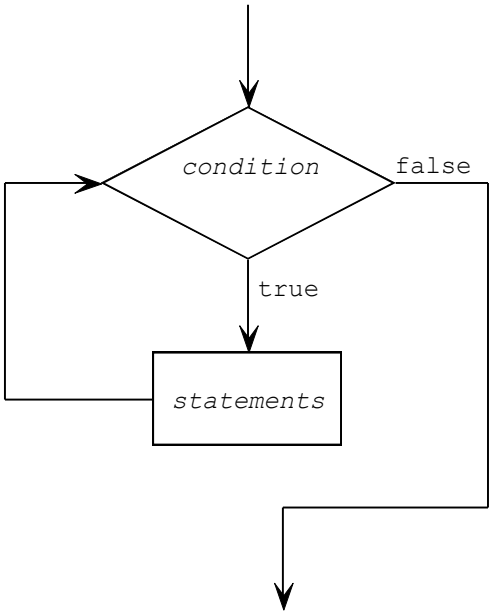
Logging is a way of tracing the execution of a program through all the method calls. More than a useful debugging tool, logging can be a business requirement in many situations. Consider that all bank transactions should be logged for accounting

purposes. However, the most common use for logging is debugging. The logging API allows a programmer to log messages at various levels of severity, capture the log to a file, and even turn logging on and off entirely.

Chapter 6: Iteration.

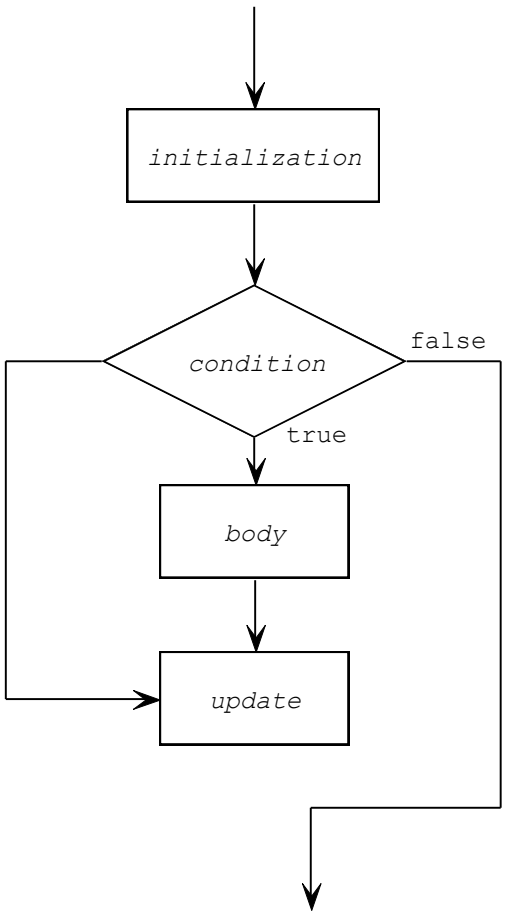
Section 6.1: `while` Loops. As mentioned in Chapter 1, computers are ideally suited for *repeatedly* performing simple tasks. Loops allow a computer program to execute the same block of code over and over so long as a condition is **`true`**. The condition is much like the condition in an **`if`** statement, but rather than executing the body once, the body continues to execute until the condition becomes **`false`**. The general syntax and structure for this construct is given below:

```
while (condition)
{
   statements
}
```



The **`while`** loop is a pre-test loop, meaning that the condition is tested *before* the execution of the code in the body of the loop. If the condition is initially **`false`**, then the loop body never executes. Further, if the condition is always true, then the loop will never terminate, resulting in an *infinite loop*.

In reality, there are four parts to every loop, but the **`while`** loop syntax only makes explicit two of the four.



The four parts are summarized in the list below.

- *Initialization*: The initialization step sets up (initializes) the variables used in the condition to their starting values. This element is inevitably outside and above the loop.

- *Condition*: The condition is the Boolean expression that determines how many times the loop body executes. This element is part of the loop and is evaluated every time the loop executes (usually at the top). When the condition becomes false due to the update, the loop will terminate.

- *Body*: The body of the loop is the block of statements that actually perform the work for which the loop was written. The body can contain all statement types including conditionals (**`if`**/**`else`**), iteration (other **`while`** loops), and sequences of straight-line code.

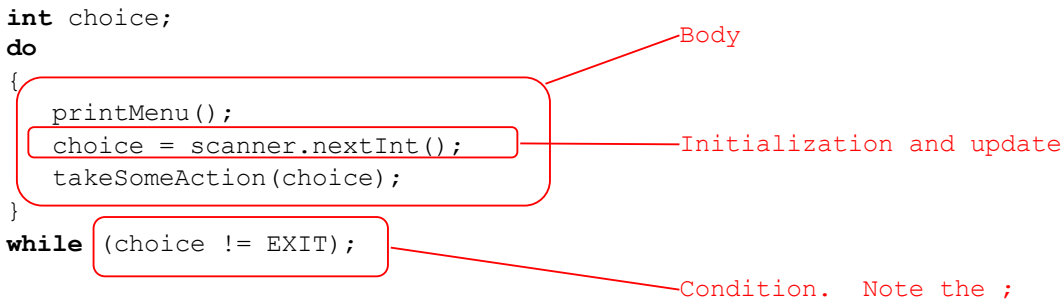
- *Update*: The update is a change to the variables in the condition that makes *progress* toward the condition becoming **`false`**. This element is in the loop and is usually one of the last statements in the body.

An error in any of the four elements will lead to particular kinds of loop problems. Debugging loops is a difficult task, but the table below summarizes the typical errors that may occur.

Problem	Description
Loop body never executes	If the loop body never executes, it is because the condition was initially false. As a result, the problem usually lies with the initialization of the condition variables or with the condition itself.
Loop never terminates	An infinite loop is the result of the condition never becoming false. The problem can exist in three places: the initialization, the condition, and/or the update. If the variables of the condition are not initialized properly, then the condition may always be true. Similarly, if the condition is written poorly, it may also always be true. Usually, though, the problem is with a bad or missing update step. If the update is missing, then no progress is made toward the condition becoming false, and therefore. the loop will never terminate. Note as well, that it is common to put a stray semicolon at the end of the while loop, making the loop body (and the update) empty! Code like this: **`while`** `(i < 10);` is almost inevitably wrong.
Loop terminates, but the number of executions is off by one.	A loop that executes one too many or one too few times is a fairly common problem. The solution is to double check both the initialization and the condition. Initializing a variable to a "too large" value could cause the loop to execute one too few times. Similarly, a condition that involves "less than" when "less than or equal" is more appropriate will halt early. Likewise, a "too small" or a "less than or equal" condition when "less than" is correct will cause the loop to execute too many times.
Loop doesn't produce the correct answer.	If the loop is terminating and executing the correct number of times, but the result is incorrect, then the problem is likely to be found in the body of the loop. The body is what does the work of the calculation, and errors there can't be diagnosed in general terms. The code is "just wrong" in the body.

Occasionally, an algorithm calls for the body of the loop to be executed *before* the condition is checked. Such a loop will always execute the body at least once. This kind of loop is known as a *post-test loop* because the condition is checked at the bottom. In Java, this is known as the **`do`**…**`while`** loop. The syntax and flow chart are given below. Though uncommon, a number of algorithms are much simpler with such a control structure. For example, a menu system printed to the screen will continue to perform selections until the user opts to exit. However, the menu

should be printed before getting user input. Pseudo-code for the algorithm using a **do**…**while** loop is shown below with the four parts of the loop highlighted.



As a general rule, **while** and **do**…**while** loops are used when the loop will execute an indeterminate number of times. When the number of iterations is known in advance, however, the **for** loop is more appropriate and is covered in the next section.

Section 6.2: `for` Loops. Semantically, there is no difference between **for** and **while** loops – everything that can be done with one can also be done with the other. However, **for** loops are perhaps easier to write because all four elements of loops (initialization, condition, body, and update) are explicit in the **for** loop syntax. The most common use of **for** loops is counting. A count-up loop initializes an integer variable (usually `i`, `j`, or `k`) to zero and continues while the value of the variable is less than some constant. At each step in the loop, the body is executed, and the update is to increment the variable. A count-down loop reverses the process. The syntax is given below:

```
for (initialization; condition; update)
{
   body
}
```

The above syntax can be easily translated into a `while` loop:

```
initialization
while (condition)
{
   body
   update
}
```

Note that the update step of a **for** loop is written at the top but is actually executed at the *bottom* of the loop. Also note that the semi-colons are part of the syntax of the **for** loop. Finally, any variable declared in the initialization element of a **for** loop is limited in scope to only the statements in the loop, allowing a programmer to reuse common counting variables in loops in the same method.

One of the most common *idioms* in programming languages derived from C (such as C++ and Java) is that counting variables *always* start with zero, and the condition *always* uses less than, but never less than or equal to. The reason behind the idiom is that arrays (covered in Chapter 7) always start their indices at 0 and the last valid index in an array of length *n* is at *n-1*. In other words, a programmer would number five items using the sequence [0, 1, 2, 3, 4]. Choosing to always start at 0, and use less than gets a programmer into the habit of thinking a particular way and dramatically reduces the number of "off by one" errors in loop processing.

Section 6.3: Nested Loops. Nested loops are loops in which one loop, called the inner loop, exists within the body of another loop, called the outer loop. A single loop is good for processing elements along a single dimension – think of counting elements on a number line. However, if you need to process information along multiple dimensions (think about rows and columns), then nested loops will be required. An example of such processing is to produce a multiplication table like the one given below.

```
 1   2   3   4   5
 2   4   6   8  10
 3   6   9  12  15
 4   8  12  16  20
 5  10  15  20  25
```

Pseudo code for this would be similar to the following:

```
function printMultTable (numRows, numCols):
   for each row i from 1 to numRows do:
      for each column j from 1 to numCols do:
         print i * j right justified in 4 spaces
      go to the next line
```

The outer loop iterates over each row in the table. The inner loop handles each column within the row. Sometimes it is easier to take the inner loop and put it in its own method. Then the body of the outer loop calls the method with the appropriate parameters, turning a nested loop into two single loops with a method call. Converting the example above would yield the following:

```
function printMultTable (numRows, numCols):
   for each row i from 1 to numRows do:
      printOneRow(i, numCols)
      go to the next line

function printOneRow (rowNumber, numCols):
      for each column j from 1 to numCols do:
         print rowNumber * j right justified in 4 spaces
```

Section 6.4: Processing Sentinel Values. In military terms, a sentinel (or sentry) is a soldier standing guard at a point. In programming, the meaning is much the same: a *sentinel* guards the end of a sequence of data to be processed. A *sentinel value* is a

value that will not normally appear in the input stream, but when received as input, terminates the algorithm. Consider a checkout line at a grocery store. The cashier begins to process items on the conveyor, scanning each item and adding up the total cost. How does the cashier know where one person's items end and another person's begins? A sentinel divider rod is used to separate one customer's items from another's items! The divider bar is a valid "input" on the conveyor but yet does not exist for sale in the store, so the cashier knows by its presence that input for one customer has ended. Pseudo-code for the cashier system follows:

```
seenSeparator = false
while not seenSeparator and conveyor.hasMoreItems() do:
   item = conveyor.nextItem()
   if item is separatorBar then:
      seenSeparator = true
   else:
      total = total + item.getPrice()
```

Notice that it wasn't possible to determine whether the loop should continue until the middle of the loop. Another way to accomplish this is to repeat the initialization and update code separately as shown below.

```
item = conveyor.nextItem()
while item is not separatorBar and
conveyor.hasMoreItems()do:
total = total+item.GetPrice()
item = conveyor.nextItem()
```

Either way, the separator bar represents a sentinel value. The first solution is a "loop and a half" style solution because determining whether or not to continue looping occurs halfway through the body of the loop. The second solution is a "prime the pump" style solution because the initialization and update are copied both within and above the loop. Choosing between the approaches is a matter of style, not correctness.

Two final tools to assist with the construction of loops are the **`break`** and **`continue`** statements. A **`break`** statement will immediately terminate the closest enclosing loop or **`switch`** statement, and control will proceed to the statement immediately outside the enclosing context. A **`continue`** statement can only be applied to loops and control immediately jumps to the bottom of the loop, where control naturally flows back to the top to have the condition tested again. Both **`break`** and **`continue`** make the control path in a program difficult to follow, so judicious use is appropriate.

Section 6.5: Random Numbers and Simulations. Simulations are models of real-world situations in which random numbers are used to simulate the occurrence of events related to the situations. For instance, to determine how many elevators are needed in a new skyscraper, a software model based on real-world data can be

constructed. Elevator speed, weight capacity, floors serviced, and even the servicing algorithm are typically pre-determined and can be built into the model. In the model, people would arrive at elevators at random (though probabilistically constrained) intervals on various floors and would travel to the lobby or vice versa. The simulation could run for 24 virtual hours and could calculate how long the average person waits for the elevator. The simulation could be run again for differing speeds and numbers of elevators in an attempt to minimize the wait time. The `Random` class uses a pseudo-random number generator to produce sequences of numbers based on parameterized constraints.

Section 6.6: Using a Debugger. A *debugger* is a program that can dynamically inspect and display information about another running program. Most debuggers are built into the integrated development environment used to edit, compile, and test programs. Many programmers start debugging their software by using logging or print messages that display the contents of variables during execution. Examining these logs shows how the state of the program changes over time and can be compared to the programmer's mental model of what *should* be happening. However, as the software gets more complex, this means of debugging quickly becomes intractable – there are too many interactions between objects to print out all the object state.

A debugger, however, allows a programmer to interact directly with a running program. There are three tools common to all debuggers: breakpoints, stepping, and inspecting. A breakpoint is a flag set on a particular line of source code that suspends the execution of the program at that line. Using the debugger interface, a programmer can then inspect all the variables that are within scope at the present point of execution. Stepping is then used to see how subsequent lines of code affect the state of the objects. Many debuggers provide several different stepping tools, including single stepping (executing the next line), step over (commonly used to execute a function in a single step), and step out (completing the current function or loop). Each debugger is different and is usually accompanied by some form of a tutorial.

Section 6.7: A Sample Debugging Session. The BlueJ debugger is quite simple. A BlueJ supplement provided by the textbook author is available at his web site. Another is available at BlueJ.org in the tutorial. The Debugging How To in the textbook is very useful, and should be read closely.

Chapter 7: Arrays and Array Lists.

Section 7.1: Arrays. Up to this point, each program has declared the individual variables it has needed in order to carry out its task, but none of these programs has needed to manipulate a large collection of data. Writing a program that tracks 1000 `Employee` objects would, until now, require declaring 1000 individual variables, likely named `employee0000`, `employee0001`, etc. If one of the desired operations was to give every employee a 2.5% cost of living adjustment, the programmer would need to write 1000 lines of code similar to `employee0000.grantRaise(0.025)`,

`employee0001.grantRaise(0.025)`, etc. What is really needed is a *single* variable that can hold 1000 `Employee` objects that can be *indexed* by an integer variable. Then the programmer could write a simple loop such as the following:

```
for (int i=0; i<1000; ++i)
{
   allEmployees[i].grantRaise(0.025);
}
```

The loop iterates over each individual element in the collection of employees called `allEmployees`. For each `Employee`, it then calls the mutator method `grantRaise` and specifies the percentage. The variable `allEmployees` is an *array* – a sequence of values of the same type.

Arrays are objects, which means that they have a type, and are created using the operator **`new`**. The valid indices of an array of length *n* are 0 through *n*-1. Each array object has a publicly accessible instance field called `length` that contains the number of elements in the array. Thus, the 1000 in the above code segment could be replaced by `allEmployees.length`. Any attempt to access an element outside of the bounds [0, `length-1`] of an array will result in an error – an `ArrayIndexOutOfBounds` exception will be thrown, and the program will halt. This is the primary reason that programmers write loops that start at zero and use asymmetric bounds, as mentioned in the reading guide notes on Section 6.2.

Arrays are usually declared and initialized at the same point in code using the operator **`new`**. The syntax for array declaration is:

```
type [] arrayName = new type[size];
```

For example, the following declares an array of type `int` and an array of type `Employee`.

```
int [] coins = new int [6];
Employee [] allEmployees = new Employee[1000];
```

When creating an array, it is important to note that each of the individual elements of the array is initialized to a default value. In the case of number types, the default value is zero; for Booleans, it is false; and for objects, it is **`null`**. In other words, creating an array of objects only creates the array, it does not create the objects within the array. The array only holds *references* to objects, not the objects themselves. For example, to populate the array of employees, code similar to the following would need to be written:


```
Employee allEmployees = new Employee[1000];
for (int i=0; i<allEmployees.length; ++i)
{
   allEmployees [i] = new Employee();
}
```

If there are fewer than 1000 employees in a company, then that number of `Employee` objects could be created, and the first **`null`** encountered in the array could act as a sentinel value for loop processing.

Section 7.2: Array Lists. Arrays are a built-in feature of the Java language. However, arrays have limitations. For instance, once declared and instantiated, an array will neither grow nor shrink. If a company keeps 1000 employees in an array and then hires the 1001st person, the code would need to be recompiled to handle the new size. Further, if the company fires employee 372, then all employees from 373 on to the end of the array would need to be "shifted" back one index in the array. Because of these limitations, the authors of Java created a library class called `ArrayList` that abstracts common operations on arrays (such as insertion and deletion of elements) into a convenient set of methods – and adds the ability to grow and shrink dynamically!

To create an `ArrayList` object, the program must import `java.util.ArrayList` and use the generics syntax to parameterize the type of data that the `ArrayList` will hold. The general syntax is:

```
ArrayList<containedType> listName = new ArrayList<containedType>();
```

The *containedType* is whatever object type the list should hold, and *listName* is the name of the variable. Using the `Employee` example from the previous section, creating a list of `Employee` would resemble:

```
ArrayList<Employee> allEmployees = new ArrayList<Employee>();
```

There are many methods for working with lists, so consult the documentation for a complete set. However, the most important for the present are `add()`, `remove()`, `get()`, `set()`, and `size()`. Quick recall on these methods and their parameters will be a time saver.

Section 7.3: Wrappers and Auto-Boxing. `ArrayList` objects are collections of other objects, which means that primitive types (number types, Boolean, and character data) cannot be directly stored in an `ArrayList`. As a result, Java provides *wrapper classes* for all primitive types. For the data type **`int`**, there exists a corresponding wrapper called `Integer`; for **`boolean`**, there is `Boolean`; for **`char`** there is `Character`; and so on. Wrapper classes merely have a data member of the wrapped type, and provide methods for interacting with it. In general, wrappers of any kind are thin veneers over an existing data type that provide a slightly different interface.

Constructing a wrapper object is relatively simple. To create an `Integer` with the value 42, construct an object using **`new`** and the constructor:

```
Integer answer = new Integer(42);
int value = answer.intValue();
```

In Java 5, however, the construction of and access to the wrapped values has been simplified by auto-boxing. Auto-boxing is the automatic construction of wrapper objects as needed, and un-boxing is accessing the internal primitive automatically as needed. Thus, in Java 5, the above code could be more easily written as:

```
Integer answer = 42;  // constructs an Integer
int value = answer;   // calls intValue
```

Thus, creating an `ArrayList` of **`int`** is simplified, as long as the contained type is Integer. The following example creates an `ArrayList` of powers of 2.

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

// note 2 vars declared & updated
for (int i=1, j=0; j<10; ++j, i*=2)
{
    powers.add(i);  // auto-box and store the int
}
```

Section 7.4: The Enhanced `for` Loop. Since it is common to create lists and arrays and then iterate over those arrays, Java 5 introduced an *enhanced* **`for`** *loop*. The enhanced **`for`** loop is used to traverse all the elements of a collection, be it an array or an `ArrayList`. The syntax is:

```
for (typeName variable : collection)
    statement
```

Applied to the `Employee` example of granting raises, this would resemble:

```
for (Employee emp : allEmployees)
{
    emp.grantRaise(0.025);
}
```

This loop would be read in English as "for each employee `emp` in `allEmployees`, grant `emp` a raise of 2.5%". In each trip through the loop, `emp` references the next element in the `allEmployees` collection. The syntax is very convenient, but narrowly scoped. The "for each" loop always starts at the beginning of the collection and it always runs until the end of the collection (unless a **`break`** statement is encountered). Using this syntax, it is not possible to start somewhere in the middle or move backwards through the collection.

Section 7.5: Simple Array Algorithms. As mentioned previously, an algorithm is a step-by-step procedure for solving a problem involving selection, iteration, and sequences of statements. Arrays or `ArrayList` data structures lend themselves naturally to a set of simple and common algorithms such as counting elements that match a criterion, finding the largest or smallest element, locating the index of a matching object, etc. Each array algorithm usually involves iterating over all elements of an array, comparing each element against some criteria, and returning a result when a match is located. Sometimes a match can be found early, but often finding a match involves scanning the entire data set (as finding a minimum/maximum does).

Given the `Employee` example again, the following method in class `Company`, which has an instance field `allEmployees`, will return a list of employees that work for a specified department:

```
public ArrayList<Employee>
findEmployeesInDepartment(Department dept)
{
    ArrayList<Employee> result =
                    new ArrayList<Employee>();
    for (Employee emp : this.allEmployees)
    {
        if (dept.equals(emp.getDepartment()))
            result.add(emp);
    }
    return result;
}
```

Interestingly, to determine how many employees work in a particular department, it would only take a single line of (sophisticated) code in class `Company`:

```
int count = findEmployeesInDepartment(
          new Department("Accounting")).size();
```

Each algorithm given in this section should be studied in detail. Chapter 19 covers additional algorithms for searching and sorting.

Section 7.6: Two-Dimensional Arrays. Not all algorithms can be implemented with a single loop; some require nested loops. Likewise, not all algorithms can be implemented with a one-dimensional array, but rather some require arrays with more dimensions. These arrays of arrays are known as multi-dimensional arrays because they use more than one index to describe a location. For example, a one-dimensional array would be similar to just one row of a spreadsheet. However, many spreadsheet computations involve several rows and columns, which is analogous to working with two-dimensional arrays. Similarly, many spreadsheet "books" have sheets as a third dimension. The number of dimensions can grow arbitrarily in computer programs, though it gets somewhat hard to visualize beyond

three (one dimension is a line, two dimensions is a plane, and three dimensions is a cube).

Declaring and creating two-dimensional arrays is similar to the process for one-dimensional arrays but uses additional brackets:

```
type arrayName [][] = new type[size1][size2];
```

One example problem that uses a two-dimensional array is a simple transposition encryption program. A user enters two numbers representing the number of rows and columns in a grid of characters. The user then enters the text to be encoded. The text is put into the grid first by row and then by column. The encrypted text is then read out first by column and then by row, effecting a transposition of the matrix. For example, using 5 rows and 8 columns, the phrase "encryption with two dimensional arrays" would be entered as:

E	N	C	R	Y	P	T	I
O	N	•	W	I	T	H	•
T	W	O	•	D	I	M	E
N	S	I	O	N	A	L	•
A	R	R	A	Y	S	•	•

The • characters replace spaces. Reading off by columns yields "`eotnannwsrc•oirrw•oayidnyptiasthml•i•e••`". To decrypt, reverse the "key" pair from (5, 8) to (8, 5), and run the same algorithm:

E	O	T	N	A
N	N	W	S	R
C	•	O	I	R
R	W	•	O	A
Y	I	D	N	Y
P	T	I	A	S
T	H	M	L	•
I	•	E	•	•

There are many additional points to this algorithm, such as padding the input with spaces to make the text length an integer multiple of the area of the matrix, flushing the matrix when it fills and then restarting at the upper left with the remaining text, etc. However, the algorithm for placing the character in the matrix and producing the result uses (not surprisingly) nested loops:

```
String plainText =
         "encryption with two dimensional arrays";
char matrix [][] = new char[rows][columns];

// **assume** that the text fits neatly
// in the matrix and encrypt
for (int r=0; r<rows; ++r)
{
    for (int c=0; c<columns; ++c)
    {
        matrix [r][c] = plainText.charAt(r*columns + c);
    }
}

// now output the encrypted text
String encryptedText = "";
for (int c=0; c<columns; ++c)
{
    for (int r=0; r<rows; ++r)
    {
        encryptedText += matrix[r][c];
    }
}
System.out.println("\"" + encryptedText + "\"");
```

This is not a particularly strong encryption algorithm, so please don't use it to encode sensitive information! However, running the encrypted text through the encryption algorithm a second time with a different set of keys makes it substantially harder to crack.

Section 7.7: Copying Arrays. An array is an object. Therefore, array variables follow object reference semantics, meaning that assigning one array reference over another makes both objects refer to the same array. In other words, assigning array variables *does not copy the data in the array*. Copying the data in the array requires allocating a new array of the same size and iterating over the elements of the array, assigning from one into the other. Here is an example of copying an array of integers:

```
int [] source = {1, 2, 3, 4, 5, 6};
int [] dest = new int[source.length];

for (int i=0; i<source.length; ++i)
    dest [i] = source[i];
```

It is also possible to clone the array using a method available to arrays, appropriately called `clone`. The above code can also be written as:

```
int [] source = {1, 2, 3, 4, 5, 6};
int [] dest = source.clone();
```

Sometimes, however, rather than copying the entire array, only a section of the array should be copied. Fortunately, copying is so common that the authors of the Java class libraries provided a method that programmers can use: `System.arraycopy()`.

As mentioned previously, an array cannot "grow" to accommodate new elements; however, a new array can be allocated that is larger than the original, and the elements can be copied from one array to the other. The original array reference can be overwritten with the new array reference, completing the "growth."

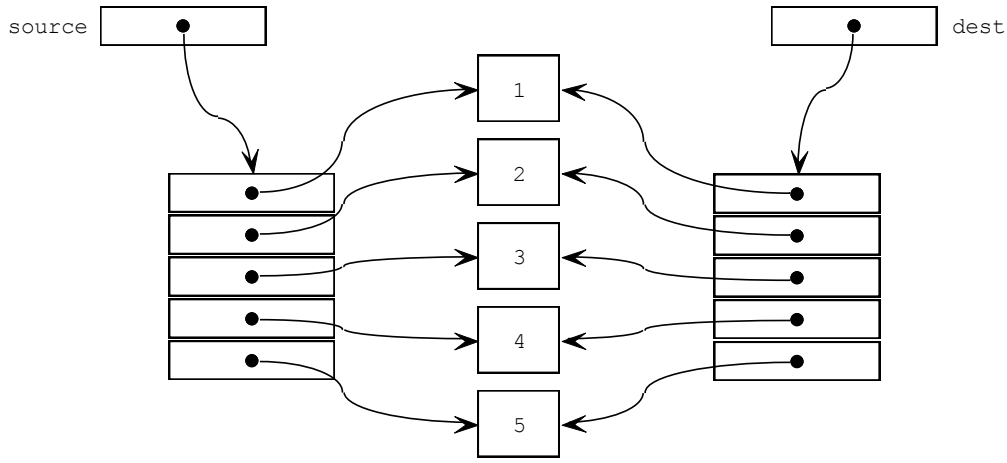
One final caveat about arrays of objects: each element of the array contains a reference. Copying references means that both arrays point to the same objects in memory. It takes more work to actually copy the objects themselves. Below is some example code, and an object diagram depicting the final state of memory

```
// where to copy from
Integer [] source = new Integer [5];

// where to copy to
Integer [] dest = new Integer[source.length];

// set up some initial data
for (int i=0; i<source.length; ++i)
    source [i] = i+1;

// copy the objects
System.arraycopy(source, 0, dest, 0, source.length);
```



It happens that `Integer` objects are immutable, so nothing "unexpected" could happen. However, consider that a programmer makes a copy of an array of `Employee` objects and then changes the name of the employee at index 0 of the first array. Since only references are copied, it also changes the name of the employee at index 0 of the second array! The `clone()` method also works by

copying references. If a true *deep copy* (where an entire structure and all its substructures are duplicated) is required, then the programmer must write the code to clone an entire *object graph*.

Section 7.8: Regression Testing. Test cases, test suites, and testing frameworks are not just useful when developing new code; they are also critical when maintaining existing code bases. Most of the software development lifecycle time is spent in maintenance – fixing errors and adding features. Over time, the code base becomes more patches than it is the original work, and the initial programming team is no longer involved. As a result, fewer and fewer people understand the complex interactions in the code, and bug fixes introduce even more bugs than they fix. At this point, the code has become *brittle*. Two practices can help with this problem: regression testing and refactoring.

Regression testing is a practice where old test cases are maintained as part of the code base and are re-run after every bug fix to ensure that the repair itself introduced no new bugs. Many programming teams require a failing test cases prior to fixing a bug so that there is a way to demonstrate that the bug was actually fixed. As more bugs are fixed, more test cases are generated, all of which can contribute to improved code quality. The second practice to prevent brittle code is refactoring. Though the term is relatively new, the concept is not: *refactoring* seeks to improve the internal design and implementation of code without affecting its externally visible behavior. Thus, it is not unusual that a "code cleanup" can proceed on a class or set of related classes that improves design and readability, yet to external consumers of the classes, no visible change occurs. To ensure that no behavior visible outside the class has changed, a regression test is run. Thus, the entire system can change, and a programmer can be confident that the system still works according to its specifications.

Chapter 14: Sorting and Searching.

Section 14.1: Selection Sort. *Sorting* is the process of ordering elements in a collection according to a particular criterion. An *ascending* sort sets the criterion such that for each element e_i of the collection, $e_{i-1} \leq e_i \leq e_{i+1}$, where i is the index of the element. A *descending* sort reverses the order such that $e_{i-1} \geq e_i \geq e_{i+1}$. There are many algorithms for sorting, the most common introductory ones being selection sort, insertion sort, and bubble sort.

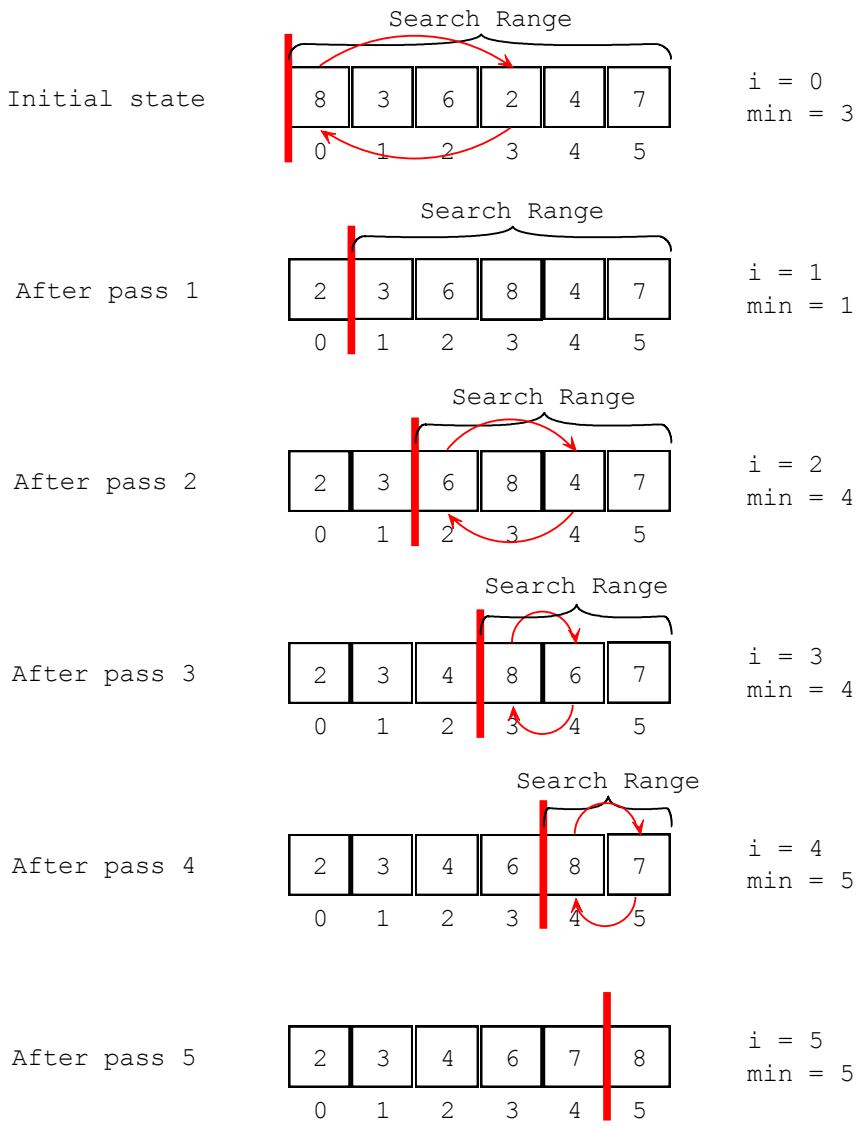
Selection sort works by dividing the array into two conceptual slices: those elements already sorted on the left of index i, and those yet to be sorted starting at i and to the right. Thus, elements $[0 \ldots i-1]$ are sorted, and $[i, n-1]$ are unsorted in an array of length n. The algorithm proceeds by finding the smallest element in the unsorted section (assume it is at index j), and then swapping the element at i with the element at j. It then increments i, and continues so long as i is less than $n - 1$ (the length of the array less 1). The code presented below is altered from that shown in the textbook to sort a subset of the array between two indices: `left` and `right`.


```
public static int [] selSort(int arr[], int left, int right)
{
    // index i separates sorted from unsorted.  At first,
    // nothing is sorted (i=left), but at the end of the
    // algorithm, everything is (i=right)
    for (int i=left; i<right; ++i)
    {
        // assume the min element is the first one to start
        int min = i;

        // locate the smallest element in the right section
        for (int j = i; j < right; ++j)
        {
            if (arr[min] > arr[j])
            {
                min = j;
            }
        }

        // swap the min element into the correct position
        int temp = arr[min];
        arr[min] = arr[i];
        arr[i] = temp;
    }
    return arr;
}
```

The illustration below shows the state of the algorithm as it proceeds through a few steps.



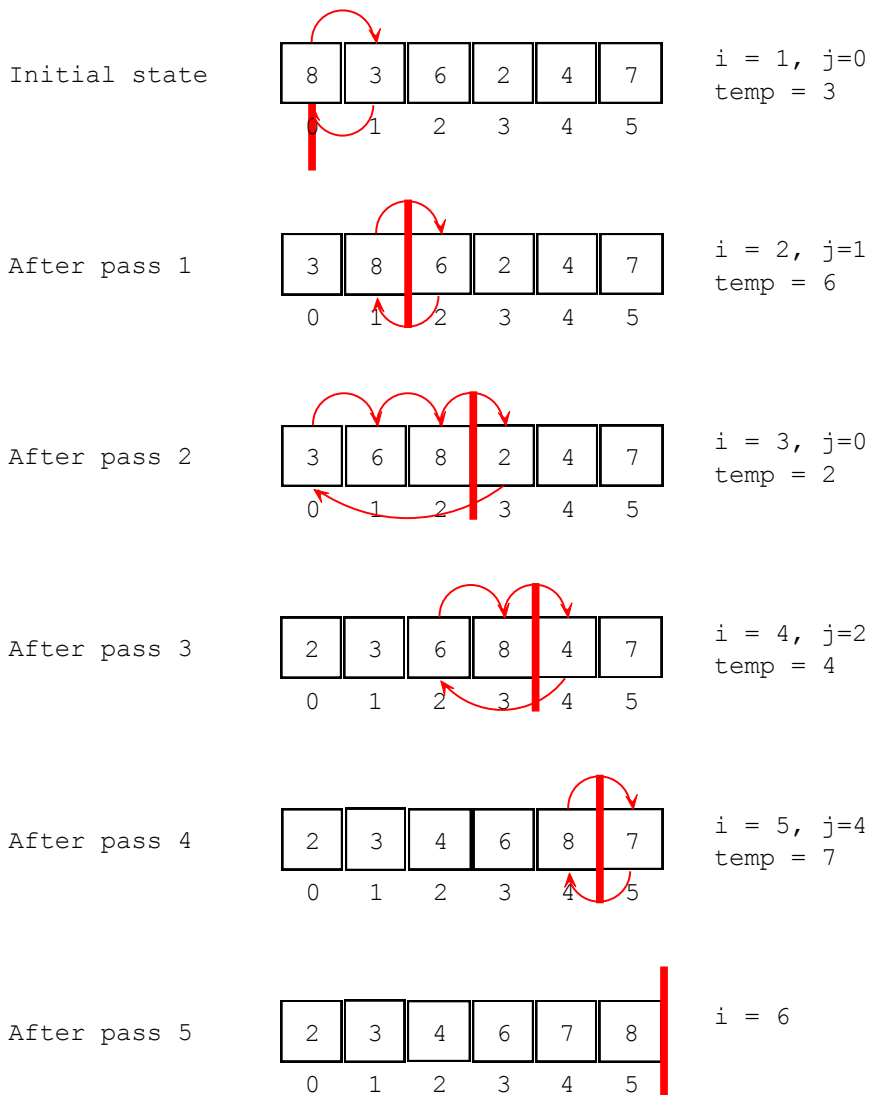
Insertion sort also works by dividing the array into sorted and unsorted slices similar to selection sort. The main difference is that insertion sort works by choosing the next unsorted element and finding where to insert it in the sorted section. When the location is found, it then shifts elements to the right and drops the element into place. More specifically, the algorithm works by selecting the element at index *i* (the first unsorted element) and copying it into a temporary variable. An index *j* starts at *i − 1* and is decremented, comparing the element at *j* against the temporary copy. If the element at *j* is greater than the temporary copy, then the element at *j* is copied to location *j + 1*. This repeats until the left index is reached, or until an element less than or equal to the temporary is found. This is the site to insert the temporary variable into the array. The code presented below is constructed to match the above selection sort and will sort a subset of the array between two indices: `left` and `right`.

```
public static int [] insSort(int arr[], int left, int right)
{
    // Note, the element at left is already sorted, it's
    // a sub-array of length 1.
    for (int i=left+1; i<right; ++i)
    {
        // make a temporary copy of the element to place
        int j, temp = arr[i];

        // Locate the position in the sorted slice to place
        // the element.  Keep within the left bound, and stop
        // when a smaller or equal element is found.  Move
        // backwards toward the beginning of the array.
        for (j=i-1; j >= left && arr[j] > val; --j)
        {
            // slide each element to the right, opening up
            // a "hole" at index j.
            arr[j+1] = arr[j];
        }

        // located the slot, drop element into position.
        arr[j+1] = temp;
    }
    return arr;
}
```

The illustration below shows the state of the algorithm as it proceeds through a few steps.



Notice that the two algorithms are complementary. What one "knows" the other "finds" and vice versa. The comparison table below summarizes this relationship:

Sort	Knows	Finds
Selection	The location to place the element	The element to place
Insertion	The element to place	The location to place the element

Finally, *bubble sort* also works by dividing the array into sorted and unsorted slices. However, in the typical implementation (shown below), the sorted slice is on the right, and the unsorted slice is on the left. There are many implementations of bubble sort, each of which adds an optimization, but all operate on the same principle: for each pair of adjacent elements in the array from left to right, compare the two elements and, if they are out of order, then swap them. At the end of one

pass through the array, it is guaranteed that one more element will have "bubbled" to the right and into its correct position. Since there are *n* elements to sort, it will take *n* -1 passes through the array. The naïve implementation of this algorithm is given below:

```
public static int[] bblSort1(int[] arr, int left, int right)
{
    // Since we're only guaranteed the correct placement
    // of one element per pass, we need n-1 passes
    for (int i=left; i<right-1; ++i)
    {
        //  Look at all the array elements per pass
        for (int j=left; j<right-1; ++j)
        {
           // if adjacent elements are out of order, swap them
           if (arr[j] > arr[j+1])
           {
                int temp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = temp;
            }
        }
    }
    return arr;
}
```

There is an easy improvement that can be made to this algorithm: don't look at every element in the array in each pass. Since bubble sort correctly places at least one element per pass (moving right to left), the inner loop can stop one element earlier with each pass. That leads to the second implementation below in which the changes have been highlighted. This algorithm is 20% faster than the previous one.

```
public static int[] bblSort2(int[] arr, int left, int right)
{
    // Since we're only guaranteed the correct placement
    // of one element per pass, we need n-1 passes
    for (int i=left, last=right-1; i<right-1; ++i, --last)
    {
        //  Look at all the unsorted elements per pass
        for (int j=left; j<last; ++j)
        {
           // if adjacent elements are out of order, swap them
           if (arr[j] > arr[j+1])
           {
                int temp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = temp;
            }
        }
    }
    return arr;
}
```

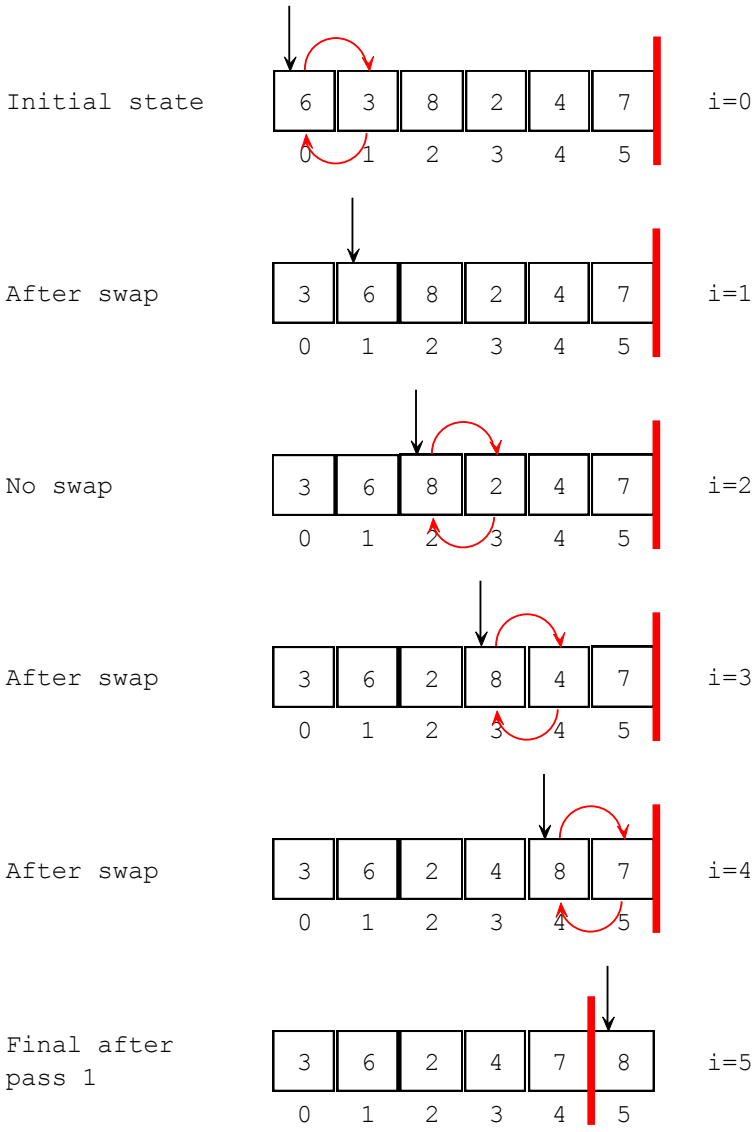
Since bubble sort not only bubbles up the largest elements, but it also tends to propagate downward the smallest elements, another improvement is possible: the array could become sorted earlier than $n - 1$ passes. At the beginning of each pass, it can be assumed that this is the last time to traverse the array. Then, when a swap actually takes place, it is known that the array wasn't really sorted to begin with, so another pass is required. In other words, the new algorithm *detects* when the array is sorted rather than requiring all $n - 1$ passes, yielding another 17% improvement over the second algorithm in cases where the data is already "mostly sorted."

```
public static int[] bblSort3(int arr[], int left, int right)
{
    boolean didSwap = true;
    int last = right-1;

    // continue as long as a swap happened in the last pass,
    // didSwap is a sentinel
    while (didSwap)
    {
        // assume the array is sorted
        didSwap = false;

        //  Look at all the unsorted elements per pass
        for (int i=left; i<last; ++i)
        {
            // detect if the array was actually unsorted
            if (arr[i] > arr[i+1])
            {
                didSwap = true;
                int temp = arr[i];
                arr[i] = arr[i+1];
                arr[i+1] = temp;
            }
        }
        --last;
    }
    return arr;
}
```

Using the final optimized algorithm above, the illustration below shows the state of the algorithm as it proceeds through a few steps. Note that there are actually two visualizations: the first shows all the swaps in the first pass through the array, and the second shows the state after each pass is complete.



Notice that, for this data, the bubble sort algorithm ended one pass earlier due to the detection of a sorted array.

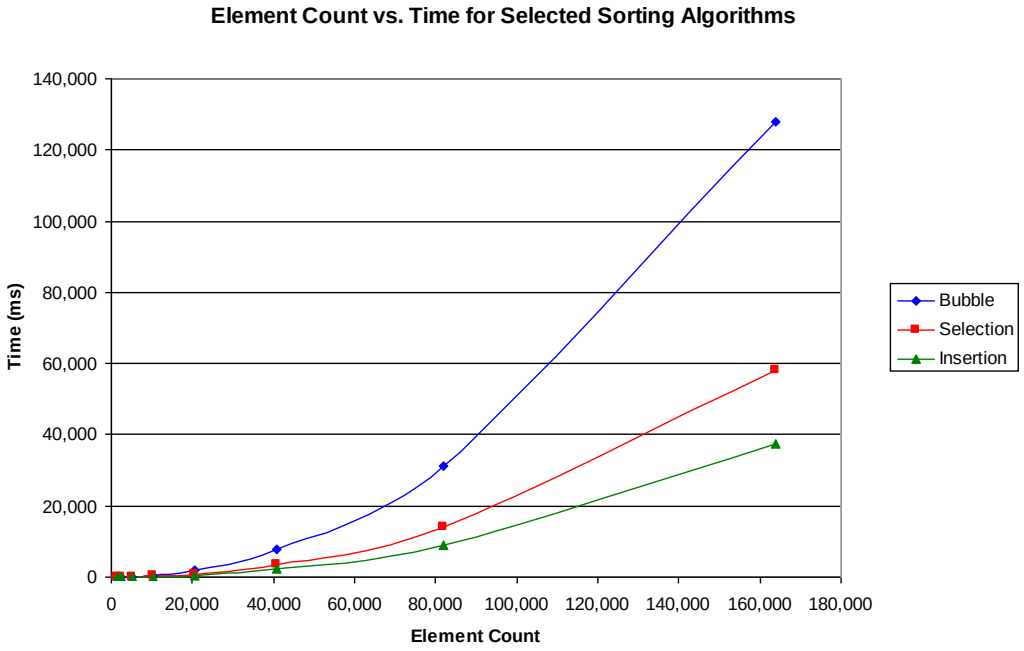
Section 14.2: Profiling the Selection Sort Algorithm. An important aspect of studying algorithms is to determine the *complexity* of a given algorithm. Complexity in computer science is usually measured two ways: *space complexity* and *time complexity*. Space complexity is a measure of the amount of memory an algorithm takes in its execution, while time complexity is a measure of the amount of time an algorithm takes in its execution. Often an inverse relationship exists: time can be reduced if space is increased, and vice-versa.

Determining the complexity of an algorithm can be done two ways: mathematical analysis (covered in Section 19.3) and *profiling*. Profiling is the process of gathering runtime statistics about programs, such as execution time, memory usage, and method call counts. In industrial environments, many tools exist for profiling programs, but for the purposes of this course, simple timing of method execution is reasonable. Timing and plotting the various sorting algorithms presented yields the following data:

Elements	Bubble	Selection	Insertion
1,280	0	0	0
2,560	31	15	15
5,120	125	47	31
10,240	484	219	156
20,480	1,969	875	563
40,960	7,829	3,516	2,313
81,920	31,391	14,047	9,125
163,840	128,078	58,000	37,375

Table 1 -- sort time in milliseconds for various array sizes.

The interesting characteristic of this data is that for each of the sorting algorithms, as the number of elements doubles, the sorting time roughly quadruples. The graph of the data below also demonstrates this effect. Section 19.3 explores this phenomenon further.



The data used to generate the data table and graph was gathered on a 2.4 GHz Pentium 4 with 512 MB memory and running Windows 2000 SP 4. It is clear from the graph that the relationship between the running time and the data size is not linear.

Section 14.3: Analyzing the Performance of the Selection Sort Algorithm. Noting the behavior of the algorithms above naturally leads to the question of why doubling the array size quadruples the run time. To understand the nature of the algorithm requires some mathematical analysis. This understanding of the

mathematics behind the code is what separates a computer scientist from a programmer. Consider this simplified version of insertion sort in which `left` is assumed to be 0, and `right` is the length of the array (i.e., the whole array is sorted):

```
 1.  public static int [] insSort(int arr[])
 2.  {
 3.      for (int i=1; i<arr.length; ++i)
 4.      {
 5.          int j, temp = arr[i];
 6.          for (j=i-1; j >= 0 && arr[j] > val; --j)
 7.          {
 8.              arr[j+1] = arr[j];
 9.          }
10.          arr[j+1] = temp;
11.      }
12.      return arr;
13.  }
```

The "innermost" piece of code is on line 8. This particular line is executed more times than any other in the code segment. What is the mathematical relationship between the number of times that code is executed and the size of the input to the method? To answer, consider the loop beginning at line 6. When $i = 1$ this loop will execute at most 1 time. When $i = 2$ the loop will execute at most 2 times. When $i =$ `arr.length - 1`, the loop will execute at most `arr.length - 1` times. The sum of all those executions is the number of times that line 8 is executed. If n is the size of the array (i.e., `arr.length`), then the expression

$$1 + 2 + 3 + \ldots + (n - 2) + (n - 1)$$

defines the number of times that line 8 is executed. What is a "nice" formula for the above sum? Reverse the sequence, and write it under the original sequence. Now, an addition down will show that all terms sum to n as shown below:

$$\begin{array}{ccccccccccc} 1 & + & 2 & + & 3 & + & \ldots & + & (n-2) & + & (n-1) \\ (n-1) & + & (n-2) & + & (n-3) & + & \ldots & + & 2 & + & 1 \\ \hline n & + & n & + & n & + & \ldots & + & n & + & n \end{array}$$

There are $(n - 1)$ the sum above is exactly double of what it should be because the series has been added together twice. So, the final formula for the number of executions of line 8 is:

$$\frac{n(n-1)}{2}$$

Multiplying out the numerator and expressing the result as a function yields the following equation:

$$f(n) = \frac{1}{2} n^2 - \frac{1}{2} n$$

When n gets to be a very large number, n^2 begins to "dominate" the result. That is, dividing by 2 or subtracting out n doesn't affect the final result very much. In other words, the graph of n^2 will always be larger than the graph of $f(n)$ that $f(n)$ in this case is $O(n^2)$ [pronounced "Big-O of n-squared"]. Any n^2 algorithm demonstrates the same properties observed in the profiling of the sort algorithm above. That is, doubling the input size (the number of elements being sorted) will quadruple the run time because:

$$\begin{aligned} f(n) &= n^2 \\ f(2n) &= (2n)^2 \\ &= 2^2 n^2 \\ &= 4n^2 \\ &= 4f(n) \end{aligned}$$

This result is good to know! Following a progression like this will let a programmer predict how long it will take to sort a large data set. For example, sorting 150,000 records above took about 2 minutes. How long would it take to sort 300 million records (about the size of the US Social Security database)? 300 million is 2000 times as large as 150 thousand, so it should take $2000^2 \times 2$ minutes or 8,000,000 minutes. That's roughly 15 *years* of sorting time! Clearly, insertion sort is an inappropriate algorithm for sorting that quantity of data. More efficient sorting algorithms will be covered in COMP 121.

Section 14.4: Merge Sort. Skip this section for this course.

Section 14.5: Analyzing the Merge Sort Algorithm. Skip this section for this course.

Section 14.6: Searching. In addition to copying, inserting into, removing from, and sorting collections of data, the last, and perhaps most commonly used, operation is searching. For arrays, searching amounts to finding the first occurrence of an element in an array, and returning its index. If the element does not exist in the array, then a negative number (an impossible index sentinel) is returned. The algorithm (modified from that shown in the textbook to search a subset of the array with asymmetrical bounds) for a linear search is given below:

```
 1.  // search a subset of an array between [left, right)
 2.  public static int search(int[ ] array, int value,
 2.      int left, int right)
 3.  {
 4.      for (int i=left; i<right; ++i)
 5.      {
 6.          if (array[i] == value)
 7.          {
 8.              return i;
 9.          }
10.      }
11.      return -1;
12.  }
```

Find the "innermost" statement in order to determine the time complexity. Clearly it is not the return statement in line 8. That is only executed once because, after execution, the function is no longer running. Actually, the innermost statement is the comparison in line 6. There are three cases to consider: the best-, worst-, and average-case execution times. The best case is when the value is found at the `left` index (the first one examined). In that case, line 6 is executed once. The worst case is when the value is not found in the array, in which case line 6 is executed (`right - left`), or n, times. The average case for when an element is found in the array is that it will take a search of roughly half the array to find an element, which is (`right - left`)/2 (or $n/2$) times. Thus, the algorithm is $O(n)$ [or linear] in its time complexity. In other words, doubling the size of the array for being searched should roughly double the time it takes to do the search using this algorithm.

Section 14.7: Binary Search. When an array is sorted, a linear search is not the most efficient means of locating an item. Consider, for instance, the game "High-Low" on "The Price is Right." In this game, the contestant has a limited amount of time to guess the cost of an expensive prize. For each guess, the host, Bob Barker, answers either "higher" or "lower" to tell the contestant that the actual retail price is above or below the guess. The algorithm for guessing a price between some high and low bounds is detailed in pseudo-code below:

```
function highLowGame(low, high):
    do:
        guess = (high+low)/2
        result = evaluateGuess(guess)

        if result is "higher" then:
            low = guess+1;
        else if result is "lower" then:
            high = guess-1;
    while result is not "correct"
```

Note that this algorithm uses symmetric bounds (i.e., low and high are "included" in the range being searched). However, programmers typically write array algorithms

to use asymmetric bounds, where the left index is included but the right index is not.

Using this algorithm, a search for a brand new 2005 Ford Explorer priced at $35,225.00, between $0.00 and $50,000.00, would proceed as follows:

Low	High	Guess	Result
0	50,000	25,000	higher
25,001	50,000	37,500	lower
25,001	37,499	31,250	higher
31,251	37,499	34,375	higher
34,376	37,499	35,937	lower
34,376	35,936	35,156	higher
35,157	35,936	35,546	lower
35,157	35,545	35,351	lower
35,157	35,350	35,253	lower
35,157	35,252	35,204	higher
35,205	35,252	35,228	lower
35,205	35,227	35,216	higher
35,217	35,227	35,222	higher
35,223	35,227	35,225	correct

Fourteen guesses to find the price is significantly better than a linear search that would take 35,226 guesses. The same technique can be applied to sorted arrays as well. The algorithm below implements a binary search on a sorted array. Note, this algorithm is adapted from the one shown in the textbook to search a subset, and the right bound is not included.

```
 1. // search a subset of an array between [left, right)
 2. public static int search(int[ ] array, int value,
 3.     int left, int right)
 4. {
 5.     // the range to search is always [left, right), which
 6.     // only contains elements if left and right haven't
 7.     // become equal or crossed over one another.
 8.     while (left < right)
 9.     {
10.         // locate the middle index between left and right
11.         int mid = (left + right) / 2;
12.
13.         if (array[mid] < value)
14.             left = mid + 1;           // in the right half
15.         else if (array[mid] > value)
16.             right = mid;              // in the left half
17.         else
18.             return mid;               // found it
19.     }
20.
21.     // not found, return "where it would have been"
22.     return -(left + 1);
23. }
```

This algorithm also returns where the element "would have been" in the array had it been located but makes the result negative to indicate it was not found.

Find the "innermost" statement in order to determine the time complexity. Clearly, the calculation of the midpoint (line 11) is executed each trip through the loop, so determining the relationship between the number of executions of line 11 to the size n of the array being searched (which is `right - left`) will give a rough idea of the time complexity. Consider a sorted array of 16,384 elements. The first pass through the loop will eliminate half of the elements under consideration, leaving 8172. The next pass cuts the number of elements to be searched down to 4096, and the next to 2048, then 1024, 512, 256, 128, 64, 32, 16, 8, 4, 2, and finally 1. Though the search could "hit" the element earlier, an unsuccessful search will always follow this pattern, which is the worst case. How many times can a sequence be divided in half? In other words, considering the current example, what is x in the equation:

$$2^x = 16384$$

To solve for x requires the use of logarithms. Just as division is the inverse operation for multiplication, and subtraction is the inverse operation for addition, logarithms are the inverse of powers. Two useful rules of logarithms are:

$$\log y^x = x \log y$$

$$\log_y x = \frac{\log x}{\log y}$$

Therefore, solving for x in the equation above gives:

$$2^x = 16384$$
$$\log 2^x = \log 16384$$
$$x \log 2 = \log 16384$$
$$x = \frac{\log 16384}{\log 2}$$

Evaluating $\log 16384 / \log 2$ yields the result 14 (verify this by using a calculator). A binary search of 16,384 elements will take at most 14 comparisons. The text shows a mathematical derivation of the above result. Without proof here (the text provides proof), binary search is $O(\log_2 n)$. Finally, back to the "High-Low" example, searching between $0.00 and $50,000.00 should take at most $\lceil \log 50000 / \log 2 \rceil = \lceil 15.609 \rceil = 16$ comparisons. (Note: The notation $\lceil \rceil$ indicates the "greatest integer" function.)

Section 14.8: Sorting Real Data. Skip this section for this course.

Chapter 8: Designing Classes.

Section 8.1: Choosing Classes. Designing classes correctly is a difficult problem with few hard and fast rules. In fact, great designs are inherently subjective even though some *metrics* – statistics about lines of code, number of objects used, lengths of parameter lists, etc. – can be gathered. Classes typically fall into one of four categories: domain classes, actors, utility classes, and library classes. Each is defined below:

- *Domain classes:* Domain classes are typically nouns extracted from the problem domain (i.e., what you're trying to solve). For example, in a payroll system, some typical domain objects would be `Employee`, `TimeCard`, and `PayCheck`. Each class represents a single concept in the problem space, and the name of the class has meaning to a non-programmer. Given just those names, a non-programmer should be able to list either what those objects do or what is done to those objects.

- *Actors:* An actor is also an element of the problem domain, but, rather than modeling a noun, this kind of class models a verb (usually a process or a sequence). The word actor in this case comes from UML – the unified modeling language – in a set of diagrams called "use cases." Back in the payroll system, example actors could be `CheckProcessor`, `FederalTaxCalculator`, and `CheckPrinter`. Note that these actors end with -er or -or suffixes as a general naming convention. Also note that, while these are in the problem domain, they do things *to* or *for* the domain classes but really don't have any data that is unique for each object.

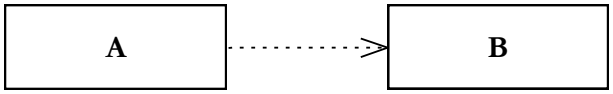
- *Utility classes:* Utility classes provide reusable services for both domain classes and actors and are often able to be used across problem domains. Utility classes are typically populated with static methods, and, therefore, no

instance of them is required. For example, the `Math` class contains only static methods and constants.

- *Library classes:* Library classes are groups of reusable components that are used to assemble a program. For instance, the `ArrayList` class studied in Section 8.2 is a library class in that instances are true objects (i.e., they have state, behavior, and identity) useful in many programs. As mentioned before, Java has a vast library of classes that handle wide varieties of tasks such as collections of data, network communications, database access, and graphical user interfaces. The API for all classes is provided in the documentation. However, it is common to use additional classes downloaded from third parties, such as the Apache Jakarta project, Codehaus, OpenSymphony, Hibernate, or Spring framework sites. Software developers also write and use their own libraries as well, and a large part of developing good software is to develop a repertoire of home-grown libraries.

Section 8.2: Cohesion and Coupling. *Cohesion* is a measure of the degree to which a class models a single concept. The public interface (public methods and constants) should all be closely related to the real-world entity being modeled. High cohesion is desirable. A program that does everything in a single class would not be cohesive, but a program that is represented by a set of collaborative objects, each of which encapsulates a single concept, would be cohesive.

Coupling is a measure of the degree to which a class depends on other classes to work properly. A dependency $A \rightarrow B$ (read as "A depends on B") exists between classes *A* and *B* if objects of class *A* use objects of class *B* as instance fields, parameters, or temporary local variables. In general, the higher the degree of coupling a class has, the more difficult the class is to develop, test, and maintain because changes to class *B* affect class *A*; therefore, low coupling is desirable. A graphical representation of the dependency relationship should be familiar from the use of the BlueJ environment, but it is also represented in UML as follows:



Minimizing coupling in object-oriented systems usually requires a design element known as *interfaces*, which is covered in COMP 121 in Section 11.1.

The `BankAccount` class developed in the text is certainly minimally coupled (as it depends on no other classes) but is not particularly cohesive. That is, the `BankAccount` class manages two concepts: account operations and money in the accounts. A better design would be to abstract out the money concept into its own class, appropriately called `Money`. This class could encapsulate internationalization (monetary symbols such as $, ¥, £, and €), adding, subtracting, comparing, and converting amounts. `BankAccount` would then be refactored to use the `Money` class. However, this change will introduce a dependency in that `BankAccount` will

now depend on the interface provided by `Money`. Thus, the concerns of coupling and cohesion are a balancing act.

Section 8.3: Accessors, Mutators, and Immutable Classes. Accessors and mutators were first mentioned in Section 2.7; however, since they are also a crucial design decision in a class, they are repeated here. Both accessor and mutators are part of the public interface. An *accessor* provides a way of extracting information from an object that has properly encapsulated its data. A *mutator*, on the other hand, alters the state of the object by changing instance fields. A class that has no mutators is called, appropriately, *immutable*. Immutable objects have some very nice properties that make them convenient for programmers, as shall be seen in Section 9.4 on side effects. The most common immutable objects are the `String` class and the wrapper classes (`Integer`, `Double`, etc.).

Section 8.4: Side Effects. A mutator changes the state of an object by altering its instance fields, and the author of a class knows which methods are mutators and which are accessors. However, consider that another programmer who uses such a mutable object may not be aware of the change. Calling a mutator method has an externally visible change on the object called a side effect. Some side effects are not expected, especially when the change happens to a parameter to a method in another class. Consider a `BankAccount` class that has a collection of `Transaction` objects within it. A third class, `StatementPrinter`, contains a method as follows:

```
 1. public class StatementPrinter
 2. {
 3.     private PrintStream out;
 4.     // much code skipped
 5.
 6.     public void print(BankAccount account)
 7.     {
 8.         List<Transaction> transactions =
 9.                 account.getTransactions();
10.         for (t : transactions)
11.             out.println(t);
12.
13.         account.archiveTransactions();
14.     }
15. }
```

The code on line 13 calls a mutator on the `BankAccount` object, which archives and then deletes the transactions for this month. A programmer calling the `print` method of a `StatementPrinter` object would suddenly have the `BankAccount` parameter changed. As a result, the programmer may need to make a *defensive copy* of the `BankAccount` object prior to calling `print`. Side effect bugs are difficult bugs to track, and this is what makes immutable objects desirable – they cannot have side effects. The rule of thumb is to minimize side effects to explicit parameters and to document all mutator methods.

Section 8.5: Preconditions and Postconditions. Preconditions and postconditions form a contract between a method and its caller. A *precondition* is any requirement about the state of an object or parameters to a method that must be met in order for the method to execute properly. The method should check its preconditions with an *assertion* before executing to prevent silent errors. A *postcondition* is a guarantee made by the method to its caller about a return value or the final state of an object after the execution of the method. Postconditions are enforced by good testing practices.

For example, consider a `transfer` method in a `BankAccount` class that moves funds from one account to another. The guarantee that this method makes is that the net amount of money in both accounts does not change and that the money is actually transferred. This guarantee is enforced via testing:

```
 1. public void testTransfer()
 2. {
 3.     // accounts with 300 and 100 dollars respectively
 4.     BankAccount from = new BankAccount(new Money(300, 0));
 5.     BankAccount to = new BankAccount(new Money(100, 0));
 6.
 7.     from.transfer(new Money(50, 0), to);
 8.
 9.     // check results
10.     assertEquals(new Money(250, 0), from.getBalance());
11.     assertEquals(new Money(150, 0), to.getBalance());
12. }
```

Lines 10 and 11 check that the postcondition – money is actually transferred – is met. In a like manner, the method itself should enforce the precondition and guarantee that the incoming objects meet its criteria. In the case of `transfer`, the precondition is that the amount of money to transfer is non-negative.

```
 1.  public class BankAccount
 2.  {
 3.      // code removed
 4.      public void transfer(Money amount, BankAccount dest)
 5.      {
 6.          assert amount.compareTo(Money.ZERO) > 0;
 7.          this.withdraw(amount);
 8.          dest.deposit(amount);
 9.      }
10.  }
```

The assert statement on line 6 checks the precondition that the transferred amount is more than zero dollars. If the condition is not met, then an `AssertionError` is "thrown" (Chapter 15) and processing stops. Note that it is the responsibility of the `withdraw` method to guarantee that the amount to withdraw does not exceed the balance, so it may have multiple assertions.

Section 8.6: Static Methods. Section 4.5 briefly addressed how to call static methods by placing the name of the class (rather than an object reference) on the left hand side of the dot operator and the name of a method on the right hand side. Contrasted to instance methods, in which the implicit parameter `this` is mapped to the object appearing on the left of the dot, static methods have no implicit parameter. Instead of "belonging" to a particular object, static methods belong to the class and would be much better named as "class methods." As such, they cannot access instance fields unless an object of that class is passed in as an explicit parameter. To write a static method, insert the **`static`** modifier after the access specifier and before the name of the method as demonstrated below:

```
public static int min(int x, int y)
{
    if (x<y)
        return x;
    return y;
}
```

When the `min` method is placed inside a utility class called `MyUtilities`, it can be called using the class name rather than an object name:

```
int smallest = MyUtilities.min(a, b);
```

Section 8.7: Static Fields. Section 4.2 briefly addressed the use of static fields when creating constants. Variable static fields are also used. However, in this case `static` does not hold the typical definition of "fixed" or "unchanging" as it relates to the value of the variable (that is what `final` means) but rather that the location in memory of this single variable is fixed. That is, all instances of an object "share" the `static` variable – it belongs to the class, not to each instance of the class.

The textbook presents an example use of static fields to create an automatically incrementing account number for bank accounts. Another common use of static fields, though, is a *design pattern* called Singleton. A design pattern is a general solution to a common problem in object-oriented programming – a problem common enough that it is written down, explained in context, and given a name. The Singleton pattern is a solution to the common problem of permitting only one instance of a class to exist in a software system – in essence, this pattern provides a single "global" object. As an example, consider that, in many business programs, it is necessary to log all critical transactions such as deposits and withdrawals in a bank account. Instead of each account having its own application log, there is one master log that all accounts use. Thus, the `withdraw` method now looks like the following:

```
1. public void withdraw(Money amount)
2. {
3.     assert amount.compareTo(Money.ZERO) > 0;
4.     assert this.balance.compareTo(amount) >= 0;
5.     Logger.getInstance().log("Withdraw " + amount
6.         + " from " + accountNumber);
7.     this.balance = this.balance.sub(amount);
8. }
```

Line 5 uses the static method `getInstance` in the `Logger` class to extract the singleton `Logger` object and log the operation. The `Logger` class is reproduced below:

```
 1. import java.util.List;
 2. import java.util.ArrayList;
 3.
 4. public class Logger
 5. {
 6.     private static Logger logger;
 7.     private List<String> allMessages;
 8.
 9.     private Logger()
10.     {
11.         allMessages = new ArrayList<String>();
12.     }
13.
14.     public static Logger getInstance()
15.     {
16.         if (logger == null)
17.         {
18.             logger = new Logger();
19.         }
20.         return logger;
21.     }
22.
23.     public void log(String message)
24.     {
25.         allMessages.add(message);
26.     }
27.
28.     public String toString()
29.     {
30.         StringBuffer buffer = new StringBuffer();
31.         for (int i=0; i<allMessages.size(); ++i)
32.         {
33.             if (i != 0)
34.                 buffer.append("\n");
35.             buffer.append(allMessages.get(i));
36.         }
37.         return buffer.toString();
38.     }
39. }
```

This is far from an industrial strength logger, but it conveys the important parts of the Singleton pattern:

- The constructor on lines 9-12 is **`private`**. This prevents the creation of `Logger` objects from outside the `Logger` class. But, how does a `Logger` get created *inside* the class?

- A *factory method* on lines 14-21 enables the creation of a `Logger` object. It is **`public`**, meaning it is accessible outside the class, and it is **`static`**, meaning that it is able to be called without a Logger instance as the implicit parameter.

- The factory method uses the **`static`** field on line 6. Since **`static`** fields belong to the class, not an instance, this is accessible from within a **`static`** method. Furthermore, the creation of the logger is delayed until some calling method actually requests it. This technique is called *lazy initialization*.

- Through the Singleton logger object all instance fields (line 7) and instance methods (lines 23-38) are accessible as well. Thus, only one log device will be used, and it can accumulate all the messages throughout the system.

Design patterns, simply stated, are the application of object-oriented techniques to yield good object-oriented designs. Patterns have been a hot topic in software development for the past several years and continue to evolve as new techniques become available. COMP 121, COMP 311, and COMP 321 will expand greatly on these concepts and will introduce many more patterns.

Section 8.8: Scope. Scope, which was mentioned briefly in the notes for Section 2.1, is the region of a program at compile time in which a variable may be accessed. A local variable is declared within a block (enclosed by braces) in a method, and its scope is from the point of declaration until the end of the block. Two local variables with overlapping scope cannot have the same name. However, two variables with disjoint scope may have the same name.

Instance members (instance fields and instance methods within a class) have a different scope than local variables. The scope of an instance member is all other instance methods. Thus, a given instance field is accessible from within all instance methods. Similarly a given instance method is also accessible from any other instance method. Since all instance methods utilize the implicit parameter **`this`**, any unqualified member access assumes that the object on which the action is taken is **`this`** object.

Static members (static fields and static methods) have a slightly altered scope. Static members are accessible from within instance methods, but, rather than being applied to **`this`** object, they are applied to the class. In other words, an unqualified static member access uses the current class by default. From within a static method, none of the instance fields or methods is accessible unless an object of that class is either created within the method or passed to the method as a parameter. In either case, instance access must be qualified by an object name on the left hand side of the dot operator.

The scopes of local variables and class and instance fields overlap. When a local variable (or parameter) has the same name as a class or instance field, then the local variable shadows (hides) the instance or class field. Using an unqualified variable name will always refer to the "closest" variable, which is the local variable or parameter. To access a shadowed class or instance field, qualify the variable by using **`this.`**`variable` or `ClassName.variable` in the case of instance and static fields respectively.

Section 8.9: Packages. As seen in Chapter 9, most well designed object-oriented programs are a tight network of collaborating objects built from a large number of small, loosely coupled, highly cohesive classes. The large number of small classes is due to the cohesion constraint. The network of collaborating objects is due to the coupling constraint. A moderately sized enterprise Java program can run to thousands of classes and tens of thousands of live objects at runtime. For instance, Tomcat version 5.5.9, a Java-based JSP and Servlet container for hosting web applications, has 1256 classes and several other external libraries with many more classes within them. As a result some organizational structure for grouping classes is needed. *Packages* group classes with similar purposes into a structure that is more manageable for programmers.

Packages correspond to the directory structure of the local storage media. For example, consider a programmer working on a product called "Alacrity" for a company called Mærlion Systems. Alacrity has a set of classes that store data into a database. The programmer, therefore, organizes the database classes into a package called `com.maerlionsystems.alacrity.persistence.manager`. If all the source code is placed under a base directory `c:\source`, then all classes in the package would be placed into `c:\source\com\maerlionsystems\alacrity\persistence\manager`. Note that the package name is the reversed Internet address to avoid potential clashes with other packages, but that is not a requirement. However, no package can begin with `java` or `javax` since those are reserved by Sun for the Java language. To create a class `JDBCPersistenceManager` within the given package, put the class in the named directory, and start the file `JDBCPersistenceManager.java` with the following declaration:

```
package com.maerlionsystems.alacrity.persistence.manager;

public class JDBCPersistenceManager
{
    // code for class goes here
}
```

Classes in the same package have direct access to other classes in the same package. However, when using classes from a different package, it is necessary to import them, or use the fully qualified class name when referencing the class. For example the following class in a different package imports and uses the `JDBCPersistenceManager` above:

```
package com.maerlionsystems.alacrity.persistence.dao;
import com.maerlionsystems.alacrity.persistence.manager.*;

public class InvoiceDao
{
    private JDBCPersistenceManager manager;
    // code for class goes here
}
```

In the example above, every class in the `com.maerlionsystems.alacrity.persistence.manager` package is imported (becomes within scope) into the current class by the use of the `*` wildcard. However, it is more common to import a single class at a time, such as **`import`** `java.util.List`.

Packages also resolve naming conflicts between classes. Two classes can have the same name as long as they are in separate packages. Either one or the other can then be imported. In the unlikely event that both classes with identical names need to be imported, the declaration and constructor calls can be fully qualified with the package name:

```
java.util.List<String> list = new java.util.ArrayList<String>();
```