

Chapter 1

A1 Introduction

page 1 of 8

Before we begin to write actual programs, we need to introduce a few basic concepts of object-oriented programming, the style of programming you will learn throughout this curriculum guide. The purpose of this lesson is to give you a feel for object-oriented programming and to introduce its conceptual foundation.

The key topics for this lesson are:

- A. [Classes and Objects](#)
- B. [Methods](#)
- C. [Objects in Software](#)
- D. [Compiling and Running a Program](#)

[A1 Vocabulary](#)

page 2 of 8

ARGUMENT	ATTRIBUTES
BEHAVIORS	CLASS
COMPILING	CONSTRUCTOR
EDITOR	INSTANCE
METHOD	OBJECT
OBJECT-ORIENTED PROGRAMMING	SOURCE CODE

A. Classes and Objects

page 3 of 8

1. Object-oriented programming (OOP) attempts to make programs more closely model the way people think about and deal with the world. In OOP, a program consists of a collection of interacting objects. To write such a program you need to describe different types of objects: what they know, how they are created, and how they interact with other objects. Each object in a program represents an item that has a job to do.

2. The world in which we live is filled with objects. For example, an object we are all familiar with is a drawing tool such as a pencil or pen. A drawing tool is an object, which can be described in terms of its attributes and behaviors. Attributes are aspects of an object that describe it, while behaviors are things that the object can do. The attributes of a pencil are its drawing color, width of the line it draws, its location on the drawing surface, etc. Anything that *describes* an object is called an attribute. Its behaviors consist of drawing a circle, drawing a line in a forward or backward direction, changing its drawing direction, etc. Anything that an object *does* is called a behavior. Another aspect of an object has to do with *creation*, which determines the initial state of an object.
3. In order to use an object within a program, we need to provide a definition for the object. This definition is called a class. The class describes how the object behaves, what kind of information it contains, and how to create objects of that type. A class can be thought of as a mold, template, or blueprint that the computer uses to create objects.
4. When building a house, a construction crew uses a blueprint to define the aspects of the house. The blueprint gives the specifications on how many bedrooms there are, how to position the electrical wiring, the size of the garage, etc. However, even two houses built from the same blueprint may have different paint colors and will have different physical locations. Clients who are buying a house may make slight modifications to these blueprints. For example, they may want a bigger garage or a smaller porch. We can see the houses built from the blueprint as objects since they are all similar in structure, but each house has its own unique attributes. In the world of programming, we can view the blueprint just like a class, i.e. a tool for creating our objects.

B. Methods

page 4 of 8

1. While a program is running, we create objects from class definitions to accomplish tasks. A task can range from drawing in a paint program, to adding numbers, to depositing money in a bank account. To instruct an object to perform a task, we send a message to it.
2. In Java, we refer to these messages as methods.
3. An object can only receive a message that it understands, which means that the

message must be defined within its class.

4. Suppose we take the `DrawingTool` class (provided by this curriculum in the package `gpdraw.jar`) and create an object `myPencil`. In OOP terminology, we say the object `myPencil` is an instance of the `DrawingTool` class. An object can only be an instance of one class. We can visually represent an object with an object diagram, as shown in Figure 1.1.

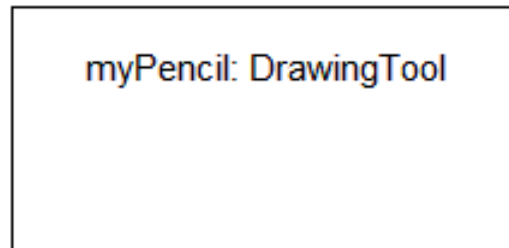


Figure 1.1 - A `DrawingTool` object named `myPencil`

5. These are some of the behaviors that the `DrawingTool` class provides:
- `forward`
 - `turnLeft`
 - `getColor`
6. To draw a line of a specified length, we call the method `forward` along with passing the distance to move the pencil. A value we pass to an object's method is called an *argument*. A diagram of calling a method is shown below in Figure 1.2.

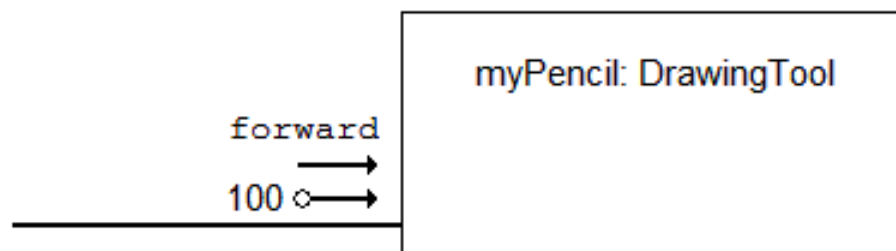


Figure 1.2 Calling the `forward` method of a `DrawingTool` object

7. If we need to change the direction `myPencil` is facing, we can call the `turnLeft` method. This will bring a ninety-degree turn to the left. Two left turns can give us a complete reversal of direction, and three left turns essentially gives us a right turn. Notice that we do not need to send any arguments with the `turnLeft` method. A left turn is simply a left turn and does not need any additional information from the user. A diagram calling `turnLeft` is shown below in Figure 1.3.

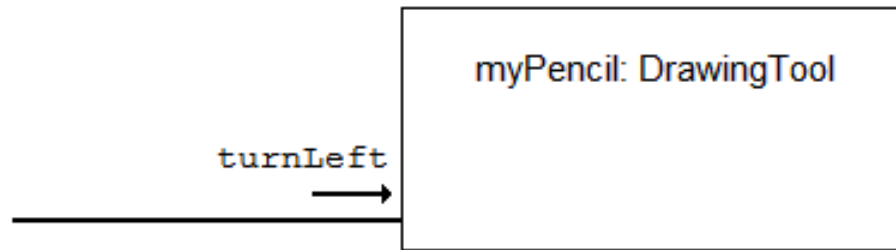


Figure 1.3 Calling the `turnLeft` method of a `DrawingTool` object

8. The diagrams shown in Figures 1.2 and 1.3 illustrate situations in which an object carries out a request by the user but does not respond to the sender. Figure 1.2 requires arguments from the user because the user must specify how far to move, whereas Figure 1.3 operates without any specific details. However, in many situations we need an object to respond by returning a value to the sender. For example, suppose we want to know the current color that is being used for drawing. We can use the `getColor` method to return the value. The `getColor` method is illustrated returning a value to the sender in Figure 1.4 below.

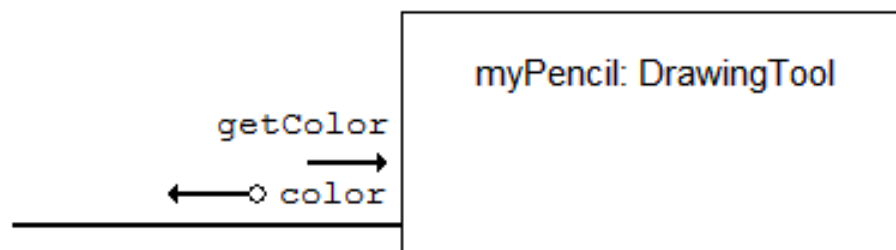


Figure 1.4 - The result of `getColor` is returned to the sender

C. Objects in Software

page 5 of 8

1. A program is a collection of instructions that performs a particular task on a computer. Software is a collection of one or more programs. Code refers to the actual symbols that a programmer types in that tell the computer what instructions to execute. Individuals who write programs are called programmers, software-engineers, software-architects, and coders among many other terms.

2. OOP is a strategy often employed by software developers. A programmer using an OOP strategy begins by selecting objects that can collectively solve the given problem.
3. To develop a particular program in an OOP fashion, the software developer might begin with a set of *program requirements*. For example:

Write a program to draw a square on a piece of paper with a pencil.

4. A way to determine the objects needed in a program is to search for the nouns of the problem. This technique suggests that the above program should have three objects: a pencil, a piece of paper, and a square.
5. Ideally, a programmer *reuses* an existing class to create objects, as opposed to writing code for a new class. For the purposes of our drawing example, we will use the preexisting `DrawingTool` and `SketchPad` classes for the pencil and paper objects. However, we don't have a class for a square that is pre-made, so we must make our own.
6. Programming languages can be compared to a foreign language - the first exposure to a written example is bound to seem pretty mysterious. You don't have to understand the details of the program shown below. They will be covered in more detail in the next lesson.

```
import gpdraw.*;
```

```
public class DrawSquare{  
    private DrawingTool myPencil;  
    private SketchPad myPaper;
```

object declarations

```
    public DrawSquare(){  
        myPaper = new SketchPad(300, 300);  
        myPencil = new DrawingTool(myPaper);  
    }
```

```
    public void draw(){  
        myPencil.forward(100);  
        myPencil.turnLeft();  
        myPencil.forward(100);  
        myPencil.turnLeft();  
        myPencil.forward(100);  
        myPencil.turnLeft();  
        myPencil.forward(100);  
    }
```

instructions

Code Sample 1.1 - `DrawSquare.java`

7. In OOP, we concern ourselves mostly with the objects themselves and how they relate to the other objects in the program. However, there must be a starting point for the program to begin creating objects, as the objects would obviously not be able to do anything if they did not exist. Code Sample 1.1 has no starting point and would therefore not be able to do anything by itself. Later on, we will learn how to utilize this code in an actual program.
8. The state of an object depends on its components. The `DrawSquare` object includes one `DrawingTool` object declared in the line that begins with the word `DrawingTool` and a `SketchPad` object declared in the line that begins with `SketchPad`. The `DrawingTool` object is given the name `myPencil` and the `SketchPad` object is given the name `myPaper`.
9. A constructor is a method with the same name as the class. The first instruction will construct a new `SketchPad` object named `myPaper` with dimensions of 300 x 300 (read as 300 by 300). The next instruction will cause a new `DrawingTool` object named `myPencil` to be constructed using the `SketchPad` object named `myPaper`.
10. An object's behavior is determined by *instructions* within its methods. When the method `draw()` for a `DrawSquare()` object is called, the instructions within the `draw` method will execute in the order they appear. There are seven instructions in the `draw` method. The first instruction will cause the `myPencil` to move forward 100 units drawing a line as it goes. The next line tells `myPencil` to turn left. The remaining 5 steps repeat the process of steps to draw the remaining three sides of the square.
11. The `DrawSquare` example illustrates the tools that a programmer uses to write a program. A program is built from objects and classes that a programmer writes or reuses. Classes are built from instructions, and these instructions are used in such a way that they manipulate objects to perform the desired tasks.

D. Compiling and Running a Program

page 6 of 8

1. A programmer writes the text of a program using a software program called an *editor*. The text of a program in a particular programming language is referred to as *source code*, or you can simply use *source* or *code* individually. The source code is stored in a file called the *source file*. For example in the `DrawSquare` example given above, source code would be created and saved in a file named `DrawSquare.java`.
2. Compiling is the process of converting a program written in a high-level language into

the *bytecode* language the Java interpreter understands. A Java compiler will generate a *bytecode file* from a source file if there are no errors in the source file. In the case of `DrawSquare`, the source statements in the `DrawSquare.java` source file would be compiled to generate the bytecode file `DrawSquare.class`. Classes inside a package, such as the *gpdrow.jar*, have already been compiled into bytecode for you.

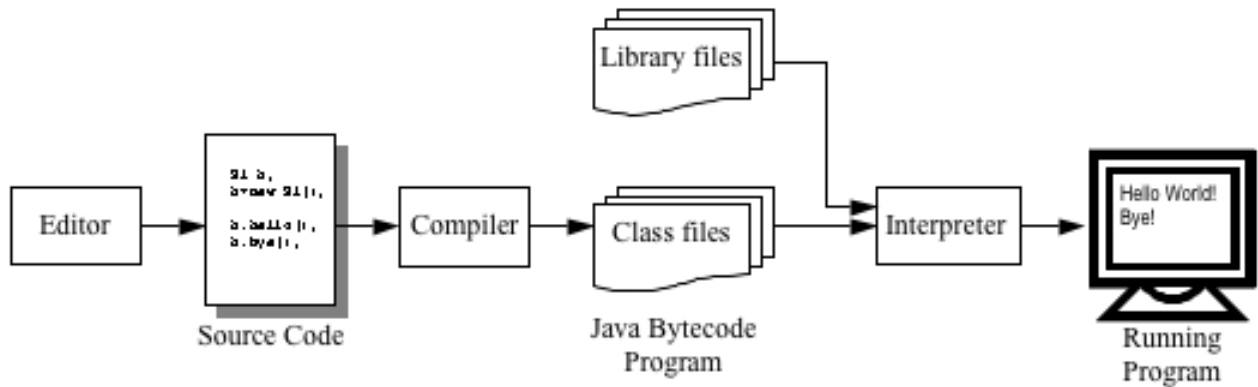


Figure 1.5 - From Source Code to Running Program

3. Errors detected by the compiler are called *compilation errors*. Compilation errors are actually the easiest type of errors to correct. Most compilation errors are due to the violation of syntax rules. These are the basic rules of languages that programmers must follow so that the interpreter understands what to do. It is similar to grammar in a spoken language and varies from language to language.
4. The Java interpreter will process the bytecode file and execute the instructions in it.
5. If an error occurs while running the program, the interpreter will catch it and stop its execution. Errors detected by the interpreter are called *run-time errors*. Run-time errors are usually caused by a fault in the logic of the program, such as accidentally causing the computer to try and divide a number by zero.

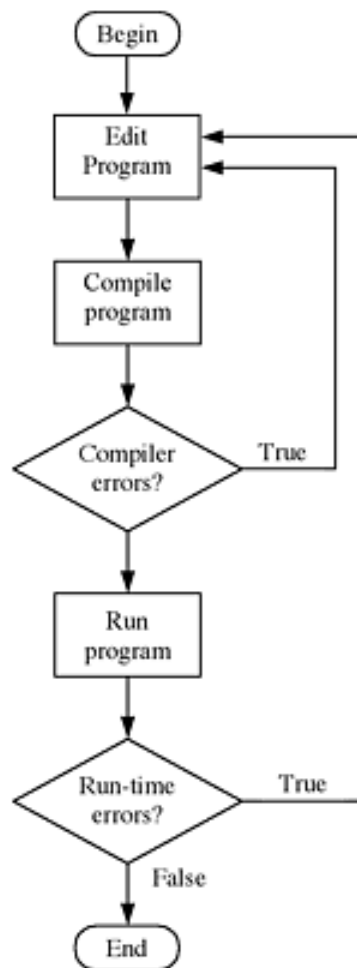


Figure 1.6 - Edit-Compile-Run Cycle for a Java Program

LAB ASSIGNMENT A1.1

page 8 of 8

DrawHouse

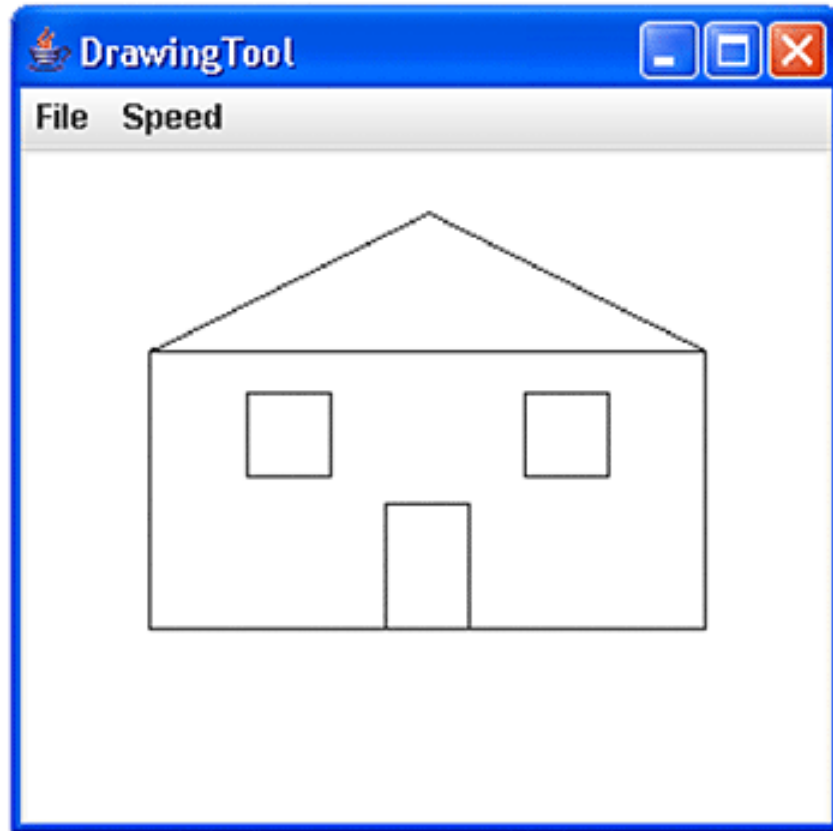
Background:

You will be provided with a file named `gpdraw.jar`, which contains the code needed to implement the graphics tools to draw objects. The specifications of the drawing tools are provided in [Handout A1.1 - DrawingTool](#). Simply place the `gpdraw.jar` file in the appropriate folder location so the Java compiler can find it. Then add this line of code at the top of your program and the drawing tools are available for use.

```
import gpdraw.*;
```

Assignment:

Write a program that creates a drawing area of appropriate size (try 500 x 500) and draws a house similar to the one shown below and with these specifications:



1. The house should fill up most of the drawing area, i.e. draw it big.
2. The house should be centered horizontally on the screen.
3. The house must have a sloped roof. It can be of any slope or configuration. But you cannot have a flat roof on the house.
4. Adding a door (centered) and windows is optional.

Instructions:

1. Include your name as a documentation statement and also a brief description of the program.
2. You will need to turn in (either on paper or electronically) a copy of your code and a picture of the house that resulted.

One can think of an OOP application as a simulated world of active objects. Each object has a set of methods that can process messages of certain types, send messages to other objects, and create new objects. Programmers can either define new classes for use in their program, or they can use pre-existing classes to create the objects for their application.

Chapter 2

A2 Introduction

page 1 of 9

Java is a “high-level” computer programming language. High-level languages are more similar to English (or other human languages) than machine code. Programming in binary (ones and zeros) or Assembly would be considered low-level. In this section, we will continue to explore the world of OOP by looking at the example from Student Lesson A1 in more detail.

The key topics for this lesson are:

- A. [Our First Java Application](#)
- B. [Program Components](#)
- C. [Object Declaration, Creation, and Message Sending](#)
- D. [Class Diagrams](#)
- E. [The Difference Between Objects and Classes](#)

[A2 Vocabulary](#)

page 2 of 9

CLASS DIAGRAM

CONSTRUCTOR

IDENTIFIER

main

new

UML

COMMENTS

DRIVER CLASS

import

MESSAGE

PACKAGE

1. Our first bit of Java code in Lesson A1 will display a square in a window as shown in Figure 2.1. Although this program is very simple, it illustrates the fundamental strategy of an object-oriented program.

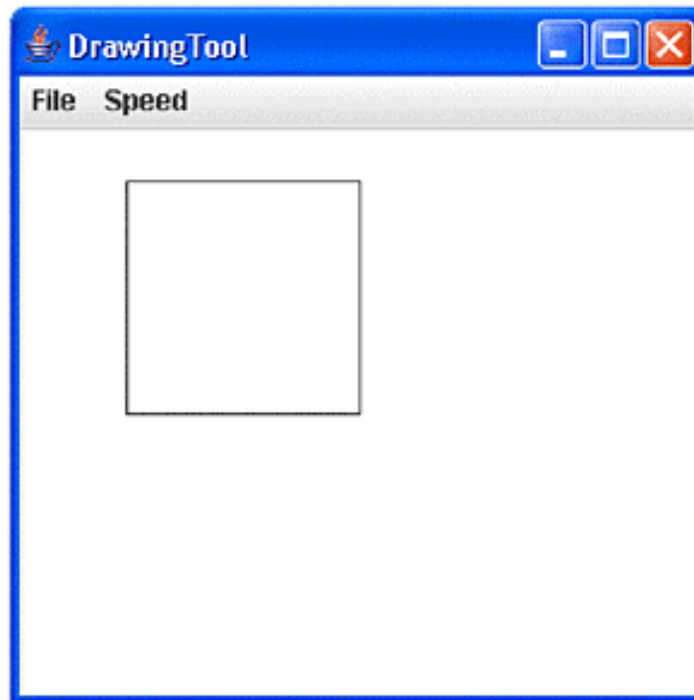


Figure 2.1 - DrawSquare

2. The following is the Java class that we encountered in the previous lesson:

```

import gpdraw.*;

public class DrawSquare{
    private DrawingTool myPencil;
    private SketchPad myPaper;

    public DrawSquare(){
        myPaper = new SketchPad(300, 300);
        myPencil= new DrawingTool(myPaper);
    }

    public void draw(){
        myPencil.forward(100);
        myPencil.turnLeft();
        myPencil.forward(100);
        myPencil.turnLeft();
        myPencil.forward(100);
        myPencil.turnLeft();
        myPencil.forward(100);
    }
}

```

Code Sample 2.1 - DrawSquare.java

3. The class is called `DrawSquare` and the class includes two methods. The first method, `DrawSquare()`, is named exactly the same as the class and is used in the construction of new instances of this class. The second method is called `draw()`, and it is where most of the action for this class takes place.
4. The `DrawSquare` constructor calls `SketchPad`'s constructor to create a new `SketchPad` object named `myPaper` with an initial size of 300 by 300 pixels. This will happen every time a new object of `DrawSquare` is created.
5. Another object called `myPencil` is created using the `DrawingTool` constructor with a drawing area represented by the `myPaper` object.
6. The method named `draw()` contains only those instructions directly related to the actual drawing of our square.
7. In order to run this program, we will create what is called a driver or throwaway class. This class serves the purpose of testing `DrawSquare`. The `DrawSquare` class should be tested and used in a small class to ensure that it is working as expected. We can then take the `DrawSquare` class and safely use it in other programs later on.

```

public class driver{
    public static void main(String[] args){
        DrawSquare sq = new DrawSquare();
        sq.draw();
    }
}

```

B. Program Components

page 4 of 9

1. Programming languages allow the inclusion of comments that are not part of the actual code. Documentation is an important aspect of writing programs as it helps the reader to understand what is going on. Java provides three different styles of comments.

```

/* Multi-line comment. This style allows you to write
   longer documentation notes as the text can wrap around to
   succeeding lines.
*/

// Single-line comment.
// This starts at the slashes and stops at the end of the line.

/** Java-doc comment. This is a special type of comment used
    to create APIs and will be discussed more thoroughly
    in a later lesson.
*/

```

2. Programmers try to avoid “reinventing the wheel” by using predefined libraries of code. In Java, these predefined classes are grouped into *packages*. The `import` statement allows the program to use predefined classes.
3. Java comes with many packages. There are also many packages created by programmers that you can use. There are two non-standard packages supplied in this curriculum guide. The one we have been using so far is called `gpdraw`. In our example program, the `DrawingTool` and `SketchPad` classes are imported from the `gpdraw` package with the statement

```
import gpdraw.*;
```

4. Every Java program contains a main method. However, not all classes need to contain a main method, nor should they.

5. Most classes will contain methods to define a class's behavior. Here we will explore the basic syntax of a method. You will learn how to utilize all of these parts in Student Lesson A4 - Object Behavior. A method has the following general syntax:

```
modifiers return_type method_name ( parameters ){  
    method_body  
}
```

- The **modifiers** refer to a sequence of terms designating different kinds of access to methods. (e.g. public, private)
- The **method_name** is the name of the method. In the case of Code Sample 2.1, the name of the method is draw.
- The **return_type** refers to the type of data a method returns. The data type can be one of the predefined types (e.g., **int**, **double**, **char**, **void**, **String**, etc.) or a user-defined type. In Code Sample 2.1, the draw method does not return a value and is therefore designated by the void type.
- The **parameters** list will allow us to send values to a method. In our example, we must tell the forward method how far to go. (myPencil.forward(100)).
- The **method_body** contains statements to accomplish the work of the method. In this example, there are seven lines of code needed to draw the square.

C. Object Declaration, Creation, and Message Sending

page 5 of 9

1. Every object in a program must be declared. An object declaration designates the name of an object and the class to which the object belongs. Its syntax is:

```
class_name object_name
```

- *class_name* is the name of the class to which these objects belong.
- *object_name* is a sequence of object names separated by commas.

In the case of the DrawSquare example, the myPencil object is declared as

```
DrawingTool myPencil;
```

other examples:

```
Account checking;  
Customer bob, betty, bill;
```

The first declaration declares an `Account` object named `checking`, and the second declares three `Customer` objects.

2. No objects are actually created by the declaration. An object declaration simply declares the name (identifier) that we use to refer to an object. Calling a constructor using the `new` operator creates an object. The syntax for **creating an object** is:

```
object_name = new class_name ( arguments ) ;
```

- `object_name` is the name of the declared object.
- `class_name` is the name of the class to which the object belongs.
- `arguments` is a sequence of zero or more values passed to the **constructor**.

In the `DrawSquare` example, the `paper` object is created (instantiated) with the statement

```
myPaper = new SketchPad(300, 300);
```

3. After the object is created, we can start sending messages to it. The syntax for sending a message to an object is

```
object_name.method_name( arguments );
```

- `object_name` is the name of the declared object.
- `method_name` is the name of a method of the object.
- `arguments` is a sequence of zero or more values passed to the method.

In the `DrawSquare` example, the `myPencil` object is sent a sequence of messages; `forward` with an argument of 100, and `turnLeft` with an argument of 90.

```
myPencil.forward(100);  
myPencil.turnLeft(90);
```

D. Class Diagrams

page 6 of 9

1. Pictures are often helpful when designing software. One particularly useful picture is the class diagram. A class diagram shows the key features of a class including:
 - the class name
 - the class attributes

- the class methods

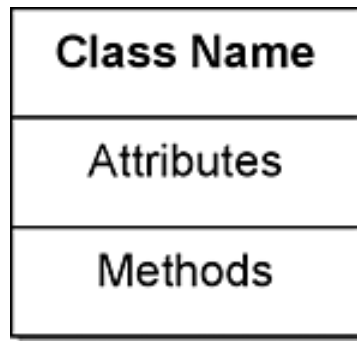


Figure 2.2 - General form of a Class diagram

2. A software class consists of attributes (think of these as nouns) and methods (think of these as verbs).
3. An attribute, or *instance variable*, represents a property of an object.
4. A method is an operation that can be performed upon an object. It is useful to picture the attributes and methods as a class diagram with the following general form.
5. The class diagram is a rectangle with three compartments separated by horizontal lines. The top compartment contains the name of the class. The middle compartment lists the attributes of the class, and the bottom compartment shows the class methods. This class notation is part of the *Unified Modeling Language (UML)*. UML is the most widely used set of notations in today's software engineering industry. A diagram for the DrawSquare class is shown below.

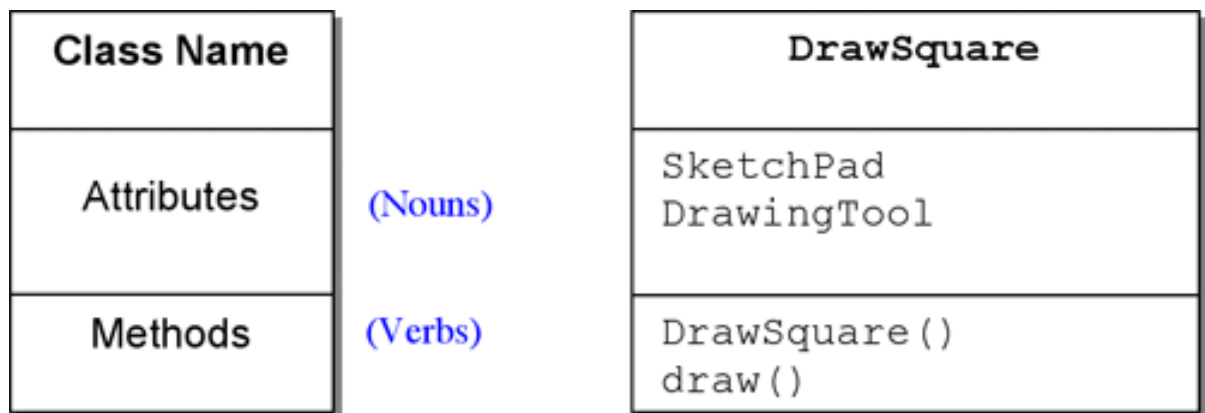


Figure 2.3 - Class diagram for the DrawSquare class

6. The methods of the class are listed in the bottom compartment of the class diagram. One of the methods in the DrawSquare class has the same name as the class (DrawSquare()). It may seem strange for a method to have the same name as its class, but this is how you give instructions on how to create objects of this class. This is called a constructor.

7. The DrawSquare class also makes use of another class: DrawingTool (the class of the myPencil object). The class diagram for this class is shown in Figure 2.4. This figure illustrates a couple of new notations that are typical of class diagrams.

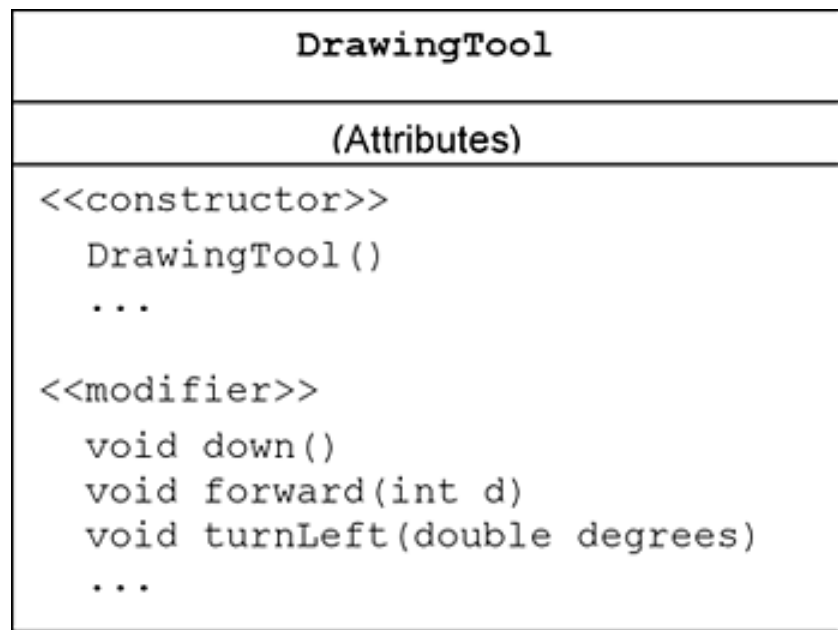


Figure 2.4 - Class diagram for the DrawSquare class

The “...” notation shown within the class diagram indicates that the list of methods is incomplete. There are more methods in the DrawingTool class that are not shown, because they are not relevant to the discussion.

UML diagrams frequently include labels within “<< >>” symbols to categorize the class methods. In Figure 2.4, the DrawingTool() method is categorized as a *constructor* method while down, forward, and turnLeft are *modifier* methods. The distinction between constructor and modifier categories will be covered in a later lesson.

E. The Difference Between Objects and Classes

page 7 of 9

1. An object is very closely associated with the class to which it belongs. An object has attributes as defined by its class. An object’s behavior is restricted by the methods that are included in its class. However, there are significant differences between objects and classes.

A class:

- is a blueprint that defines attributes and methods.
- is written by a programmer.
- cannot be altered during program execution.
- is named by a class name.

An object:

- must belong to some class.
 - is an instance of that class.
 - exists during the time that a program executes.
 - must be explicitly declared and constructed by the executing program.
 - has attributes that can change in value and methods that can execute during program execution. (The class to which the object belongs defines these attributes and methods.)
 - is most often referenced using an identifier.
2. Classes can be compared to a blueprint. The purpose of a blueprint is to provide a guide for creating things, just as the purpose of a class is to guide the creation of objects. A single blueprint is designed to produce a single basic type of object. Similarly, the objects from the same class all share common characteristics.
 3. Each object must belong to one particular class, and the object is said to be a member of the class to which it belongs. The `myPencil` object that was created earlier belongs to the `DrawingTool` class. This means that the `myPencil` object is permitted to perform `DrawingTool` methods (operations). This also means that the `myPencil` object is *not* permitted to perform `DrawSquare` methods; these methods are designed for a different kind (class) of object.

Summary/Review

page 8 of 9

A programmer designs and writes classes that can later be used in other classes to create objects of that class. The objects then work together to solve your problem. Breaking down a large problem into a collection of smaller problems in this way is a very useful programming technique. When designing a class, we must consider the behaviors and attributes of the objects created from that class, as well as how we will create those new objects.

Chapter 3

When designing a class, one of the programmer's jobs is to define the attributes of the class. Java allows two different types of attributes: objects and primitive data types. Using objects as attributes will come naturally as you become more used to OOP. Java provides several primitive data types to store basic information and uses objects to fill in gaps that are not provided with the primitive data types. Java is a richly typed language, which means that it gives the programmer a wide variety of data types to use. As the name suggests, primitive data types are very basic in nature. In this lesson you will declare variables, store values in them, learn operations to manipulate and use those values, and print out the values using the `System.out` object.

The key topics for this lesson are:

- A. [Identifiers in Java](#)
- B. [Primitive Data Types in Java](#)
- C. [Declaring and Initializing Variables in Java](#)
- D. [Printing Variables Using the System.out Object](#)
- E. [Assignment Statements](#)
- F. [Math Operators](#)
- G. [Precedence of Math Operators](#)
- H. [Assignment Operators](#)
- I. [Increment and Decrement Operators](#)

[A3 Vocabulary](#)

ASCII	ASSIGNMENT OPERATOR
boolean	char
DECREMENT OPERATOR	double
ESCAPE SEQUENCE	float
INCREMENT OPERATOR	IDENTIFIER
int	MODULUS OPERATOR
PRECEDENCE	PRIMITIVE DATA TYPE
RESERVED WORDS	STRING LITERAL

A. Identifiers in Java

1. An identifier is a name that will be used to describe classes, methods, constants, variables; anything a programmer is required to define.
2. The rules for naming identifiers in Java are:
 - Identifiers must begin with a letter.
 - Only letters, digits, or an underscore may follow the initial letter.
 - The blank space cannot be used.
 - Identifiers cannot be reserved words. Reserved words or keywords are already defined in Java. These include words such as *new*, *class*, *int*, etc. **See [Handout A3.1](#), Reserved Words in Java**

Java is a case sensitive language. That is, Java will distinguish between upper and lower case letters in identifiers. Therefore:

`grade` and `Grade` are different identifiers

3. Be careful both when naming identifiers and when typing them into the code. Be consistent and don't use both upper and lower case names for the same identifier.
4. A good identifier should help describe the nature or purpose of whatever it is naming. For a variable name, it is better to use

`grade` instead of `g`, `number` instead of `n`.

5. However, avoid excessively long or "cute" identifiers such as:

`gradePointAverageForStudentsAToZ`
Or `bigHugeUglyNumberThatIsPrettyPointlessButINeedItAnyway`

Remember that the goal is to write code that is professional in nature; other programmers need to understand your code.

6. Programmers will adopt different styles of using upper and lower case letters in writing identifiers. The reserved keywords in Java must be typed in lower case text, but identifiers can be typed using any combination of upper and lower case letters.

7. The following conventions will be used throughout this curriculum guide:

- A single word identifier will be written in lower case only. Examples: grade, number, sum.
- Class names will begin with upper case. Examples: String, DrawingTool, SketchPad, Benzene.
- If an identifier is made up of several words, all words beyond the first will begin with upper case. Examples: stringType, passingScore, largestNum, DrawHouse, SketchPad.
- Identifiers used as constants will be fully capitalized. Examples: PI, MAXSTRLEN.

B. Primitive Data Types in Java

page 4 of 14

1. Java provides eight primitive data types: **byte**, **short**, **int**, **long**, **float**, **double**, **char**, and **boolean**. The data types **byte**, **short**, **int**, and **long** are for integers, and the data types **float** and **double** are for real numbers (numbers with decimal places).
2. The College Board Advanced Placement (AP) Examinations only require you to know about the **int**, **double**, and **boolean** data types. This curriculum will, from time to time, also use the **char** type when appropriate.
3. An integer is any positive or negative number without a decimal point.

Examples: 7 -2 0 2025

4. A double is any signed or unsigned number with a decimal point. Doubles cannot contain a comma or any symbols. A double value can be written using scientific notation.

- Valid numbers: 7.5 -66.72 0.125 5
- Invalid numbers: \$37,582.00 #5.0 10.72%
- Scientific notation: 1625. = 1.625e3 .00125 = 1.25e-4

(Note: When applying 5 to a double variable, Java will automatically add the decimal point for you.)

5. The following table summarizes the bytes allocated and the resulting size.

Size	Minimum Value	Maximum Value
------	---------------	---------------

byte	1 byte	-128	127
short	2 bytes	-32768	32767
int	4 bytes	-2147483648	2147483647
long	8 bytes	-9223372036854775808	9223372036854775807
float	4 bytes	-3.40282347E+38	3.40282347E+38
double	8 bytes	-1.79769313486231570E+308	1.79769313486231570E+308

6. Character type consists of letters, digits 0 through 9, and punctuation symbols. A character must be enclosed within single quotes.

Examples: 'A', 'a', '8', '*'

Java characters are stored using 2 bytes according to the ASCII code. ASCII stands for American Standard Code for Information Interchange.

See [Handout A3.2, ASCII Characters - A Partial List](#)

Using ASCII, the character value 'A' is actually stored as the integer value 65. Because a capital 'A' and the integer 65 are physically stored in the same fashion, this will allow us to easily convert from character to integer types, and vice versa.

```
char letter = 'A';
int number = 75;
System.out.println("letter = " + letter);

System.out.print("its ASCII value = ");
System.out.println((int)letter);

System.out.print("ASCII value 75 = ");
System.out.println((char)number);
```

Run output:

```
letter = A
its ASCII value = 65

ASCII value 75 = K
```

7. In Java, you can make a direct assignment of a character value to an integer variable, and vice versa. This is possible because both an integer and a character variable are

ultimately stored in binary. However, it is better to be more explicit about such conversions by using type conversions. For example, the two lines of code below assign to position the ASCII value of letter.

```
char letter = 'C'; // ASCII value = 67
int position;

position = letter;
//This is legal, position now equals 67
//However, another programmer might think this is an
//accident.

vs.

position = (int)letter;
//Here it is immediately clear what you mean to do.
//This is called self-documenting because the code shows
//what the meaning is without the need for comments.
```

8. Programmers use the single quote to represent char data and double quotes to denote String types. To use either of those characters in a String literal, such as in a `System.out.println` statement, you must use an escape sequence. Java provides escape sequences for unusual keystrokes on the keyboard. Here is a partial list:

Character	Java Escape Sequence
Newline	'\n'
Horizontal tab	'\t'
Backslash	'\\'
Single quote	'\''
Double quote	'\"'
Null character	'\0'

```
System.out.println("This is a\ntest and only\' a test.");
```

Run output:

```
This is a
test and only' a test.
```

9. Data types are provided by high-level languages to minimize memory usage and processing time. Integers and characters require less memory and are easier to process. Floating-point values require more memory and time to process.
10. The last primitive data type is the type `boolean`. It is used to represent a single `true/false` value. A `boolean` value can have only one of two values:

true

false

In a Java program, the reserved words **true** and **false** always refer to these **boolean** values.

C. Declaring and Initializing Variables in Java

page 5 of 14

1. A variable must be declared before it can be initialized with a value. The general syntax of variable declarations is:

```
data_type variableName;
```

for example:

```
int number;  
char ch;
```

2. Variables can be declared in a class outside of any methods or inside of a method. Variables can also be declared and initialized in one line. The following example code illustrates these aspects of variable declaration and initialization.

```
// first is declared and initialized  
// second is just declared  
int first = 5, second;  
double x;  
char ch;  
boolean done;  
  
second = 7;  
x = 2.5;  
ch = 'T';  
done = false;  
  
int sum = first + second;
```

Code Sample 3-1

Note that multiple variables can be declared on one line. Initialization is done using the assignment operator (=). Initialization can occur at declaration time or later in the program. The variable `sum` was declared and used in the same line.

3. Where variables are declared is a matter of programming style and need since this determines how and where they can be used.

1. The `System.out` object is automatically created in every Java program. It has methods for displaying text strings and numbers in plain text format on the system display, which is sometimes referred to as the “console.” For example:

```
int    number = 5;
char   letter = 'E';
double average = 3.95;
boolean done = false;

System.out.println("number = " + number);
System.out.println("letter = " + letter);
System.out.println("average = " + average);
System.out.println("done = " + done);
System.out.print("The ");
System.out.println("End!");
```

Run output:

```
number = 5
letter = E
average = 3.95
done = false
The End!
```

Code Sample 3-2

2. Method `System.out.println` displays (or prints) a line of text in the console window. When `System.out.println` completes its task, it automatically positions the output cursor (the location where the next character will be displayed) to the beginning of the next line in the console window (this is similar to pressing the *Enter* key when typing in a text editor-the cursor is repositioned at the beginning of the next line).

3. The expression

`"number = " + number`

from the statement

```
System.out.println("number = " + number);
```

uses the `+` operator to “add” a string (the literal `"number = "`) and `number` (the

int variable containing the number 5). Java defines a version of the + operator for *String concatenation* that enables a string and a value of another data type to be concatenated (added). The result of this operation is a new (and normally longer) String. String concatenation is discussed in more detail later on.

4. The lines

```
System.out.print("The ");  
System.out.println("End!");
```

of Code Sample 3-2 display one line in the console window. The first statement uses System.out's method, print, to display a string. Unlike println, print does not position the output cursor at the beginning of the next line in the console window after displaying its argument. The next character displayed in the console window appears immediately after the last character displayed with print.

5. Note the distinction between sending a String literal, "number = ", versus a variable, number, to the System.out object. A **boolean** variable will be printed as **true** or **false**.
6. The result (or output) of formulas using Strings may also be printed. Note how the placement of the quotes affects the output.

```
System.out.println( 2 + 2);  
//Output: 4  
System.out.println("2 + 2");  
//Output: 2 + 2  
System.out.println("2" + "2");  
//Output: 22
```

E. Assignment Statements

page 7 of 14

1. An assignment statement has the following basic syntax:

```
variable = expression;
```

The assignment operator (=) assigns the value of the expression on the right to the variable.

```
a = 5;
```

This is not the same as saying, “a equals five,” but is more akin to, “a receives the value five.”

The expression can be a literal constant value such as 2, 12.25, 't' or it can also be a numeric expression involving operands (variables or constants) and operators.

```
a = 5 + 2;  
b = 6 * a;
```

The assignment operator returns the value of the expression. Returning values in this way allows for chaining of assignment operators. Chaining is when you have more than one assignment in a statement as shown below.

```
a = b = 5;
```

The assignment operator is right-associative. This means that the above statement is really solved in this order:

```
a = (b = 5); // solved from right to left.
```

Since (b = 5) returns the integer 5, the value 5 is also assigned to variable a.

2. Variables contain either primitive data or object references. Notice that there is a difference between the two statements:

```
primitiveValue = 18234;
```

and

```
myPencil = new DrawingTool();
```

A variable will *never* actually contain an object, only a reference to an object. In the first statement, `primitiveValue` is a primitive type, so the assignment statement puts the data directly into it. In the second statement, `myPencil` is an object reference variable (the only other possibility) so a reference to the object is put into that variable. The object reference tells the program where to find an object.

Variable Type	Information It Contains	When On the Left of "="
primitive	Contains actual data	Previous data is replaced with new data
object	Contains a reference,	Old reference is

	i.e. information on how to find the object referred to by the variable	replaced with a new reference
--	--	-------------------------------

3. The two types of variables are distinguished by how they are declared. Unless it was declared to be of a primitive type, it is an object reference variable. A variable will not change its declared type.
4. Be aware. Because these assignments work differently with primitive data types and with objects, you may experience unexpected behavior. Consider the following:

```
DrawingTool pencil = new DrawingTool(300,300);
pencil.setColor(Color.red);
DrawingTool pen = pencil;
pen.setColor(Color.blue);
pencil.forward(100);
```

In this example, pencil will draw a blue line instead of red. This happens because we have only created one object here; we only have one new statement. When we say `DrawingTool pen = pencil`, all we are doing is assigning the variable name of 'pen' to the same object that pencil is already assigned to. This means that both pen and pencil refer to the same `DrawingTool` object, and what you tell pen to do is also happening to the pencil, because they are the same object. This may be confusing at first, but once you begin to utilize OOP, it will begin to make more sense.

F. Math Operators

page 8 of 14

1. Java provides 5 math operators as listed below:

- + Addition, as well as unary +
- Subtraction, as well as unary -
- * Multiplication
- / Floating point and integer division
- % Modulus, remainder of integer or floating point

division

- The numerical result and data type of the answer depends on the type of operands used in a problem.
- For all the operators, if both operands are integers, the result is an integer. Examples:

2 + 3 -> 5
4 * 8 -> 32

9 - 3 -> 6
11/2 -> 5

Notice that 11/2 is 5, and not 5.5. This is because ints work *only* with whole numbers. The remaining half is lost in integer division.

- If either of the operands is a double type, the result is a double type. Examples:

2 + 3.000 -> 5.000
25 / 6.75 -> 3.7037
11.0 / 2.0 -> 5.5

When an integer and a double are used in a binary math expression, the integer is promoted to a double value, and then the math is executed. In the example `2 + 3.000 -> 5.000`, the integer value 2 is promoted to a double (2.000) and then added to the 3.000.

- The modulus operator (%) returns the remainder of dividing the first operand by the second. For example:

10 % 3 -> 1
16 % 2 -> 0

2 % 4 -> 2
27.475 % 7.22 -> 5.815

- Changing the sign of a value can be accomplished with the negation operator (-), often called the unary (-) operator. A unary operator works with only one value. Applying the negation operator to an integer returns an integer, while applying it to a double returns a double value. For example:

-(67) -> -67

-(-2.345) -> 2.345

- To obtain the answer of 5.5 to a question like 11/2, we must cast one of the operands.

(double)11/2

results in 5.5

The casting operators are unary operators with the following syntax:

(**type**) operand

The same effect can also result from simply

G. Precedence of Math Operators

1. Precedence rules govern the order in which an expression is solved. For example:

$$2 + 3 * 6 \rightarrow 20 \quad \text{the } * \text{ operator has priority over } +.$$

2. Associativity refers to the order in which operators are applied if they have the same precedence level. The two possibilities are from left-to-right or right-to-left.
3. A unary operator is used on only one number. An example of a unary operator is the negative sign in the expression $-a$, meaning the negative of a .
4. The following table summarizes precedence and associativity of math operators:

Level of Precedence	Operator	Associativity
Highest	unary -	right to left
	* / %	left to right
Lowest	+ -	left to right

5. An example follows:

$9 + 16 / 3 * 7 \% 8 - 5$	(solve / first)
$9 + 5 * 7 \% 8 - 5$	(solve * second)
$9 + 35 \% 8 - 5$	(solve % next)
$9 + 3 - 5$	(solve left-to-right)
7	

6. Parentheses take priority over all the math operators.

$$(5+6)/(9-7) \rightarrow 11/2 \rightarrow 5$$

(integer division, which drops remainders, is used here)

1. The statement `number = number + 5;` is an example of an accumulation statement. The old value of `number` is incremented by 5 and the new value is stored in `number`.
2. The above statement can be replaced as follows:

```
number += 5;
```

3. Java provides the following assignment operators:

```
+=    -=    *=    /=    %=
```

These statements are preferable to saying `number = number + 5` because they are more convenient and easier to read. You can immediately tell at a glance exactly what is being done.

4. The following examples are equivalent statements:

<code>rate *= 1.05;</code>	<code>rate = rate * 1.05;</code>
<code>sum += 25;</code>	<code>sum = sum + 25;</code>
<code>number %= 5;</code>	<code>number = number % 5;</code>

5. The precedence of the assignment operators is the lowest of all operators.

1. Incrementing or decrementing by one is a common task in programs. This task can be accomplished by the statements:

```
n = n + 1;   or   n += 1;
```

2. Java also provides a unary operator called an increment operator, `++`.
3. The statement `n = n + 1` can be rewritten as `++n`. The following statements are equivalent:

```
n = n + 1;           ++n;
```

```
sum = sum + 1;
```

```
++sum;
```

4. Java also provides for a decrement operator, --, which decrements a value by one. The following are equivalent statements:

```
n = n - 1;
```

```
--n;
```

```
sum = sum - 1;
```

```
--sum;
```

5. The increment and decrement operators can be written as either a prefix or postfix unary operator. If the ++ is placed before the variable it is called a pre-increment operator (++number), but it can follow after the variable (number++), which is called a post-increment operator. The following three statements have the same effect:

```
++number;    number++;    number = number + 1;
```

6. Before looking at the difference between prefix and postfix unary operators, it is important to remember Java operators solve problems and often return values. Just as the assignment operator (=) returns a value, the ++ and -- operators return values. Consider the following code fragments:

```
int a=1, b;  
b = ++a;
```

```
int a=1, b;  
b = a++;
```

After execution of the above code
a = 2 and b = 2

After execution of the above code
a = 2 and b = 1

7. The statement `b = ++a` uses the pre-increment operator. It increments the value of `a` and returns the new value of `a`.
8. The statement `b = a++` uses the post-increment operator. It returns the value of `a` and then increments `a` by 1.
9. The precedence and associativity of the unary increment and decrement operators is the same as the unary - operator.

Some cash register systems use change machines that automatically dispense coins. This lab will investigate the problem solving and programming behind such machinery. You always want to use the fewest coins possible. You should use integer mathematics to solve this problem.

Provide the number of cents through the constructor. Write a method that calculates the number of each type of coin.

Examples :

```
35 cents =>
Quarter(s)   1
Dime(s)      1
Nickel(s)    0
Penny(s)     0
```

```
41 cents =>
Quarter(s)   1
Dime(s)      1
Nickel(s)    1
Penny(s)     1
```

Assignment:

1. Follow the same format that was used in Lab Assignment A3.1 (Easter), using a driver and a class called Coins.
2. Run the samples from above to check your work.
3. Run the following three samples and copy the sample runs into your class file, print out the code for the class and hand in.

```
94 cents
59 cents
19 cents
```

4. Do not worry about singular versus plural endings, i.e. quarter/quarters.

Summary/Review

page 12 of 14

This lesson has covered a great amount of detail regarding the Java language. At first, it is necessary to memorize the syntax of data types and their operations, but with time and

practice, fluency will come. As classes are designed and code is written to solve problems, a primitive data type will often be chosen to store basic information.

Chapter 4

A4 Introduction

page 1 of 11

It was recognized long ago that programming is best accomplished by working with smaller sections of code that are connected in very specific and formal ways. Programs of any significant size should be broken down into smaller pieces. Classes can be used to create objects that will solve those smaller pieces. We determine what behaviors these objects will perform. These behaviors of the objects are called methods.

The key topics for this lesson are:

- A. [Writing Methods in Java](#)
- B. [Parameters and Arguments](#)
- C. [Returning Values](#)
- D. [The Signature of a Method](#)
- E. [Lifetime, Initialization, and Scope of Variables](#)
- F. [Getters and Setters](#)

[A4 Vocabulary](#)

page 2 of 11

ARGUMENT	GETTERS
METHOD	PARAMETER
return	SCOPE
SETTERS	SIGNATURE

1. Methods are what an object can actually do, such as in our DrawingTool example:

```
myPencil.forward(100);  
myPencil.turnLeft();
```

2. Revisiting our example from Student Lesson A2, we can see that we have already been using methods.

```
import gpdraw.*;  
  
public class DrawSquare{  
  
    private DrawingTool myPencil;  
    private SketchPad myPaper;  
  
    public DrawSquare(){  
        myPaper = new SketchPad(300, 300);  
        myPencil= new DrawingTool(myPaper);  
    }  
  
    public void draw(){  
        myPencil.forward(100);  
        myPencil.turnLeft();  
        myPencil.forward(100);  
        myPencil.turnLeft();  
        myPencil.forward(100);  
        myPencil.turnLeft();  
        myPencil.forward(100);  
    }  
}
```

Code Sample 4-1

We wrote our own methods (DrawSquare, draw) and we used some methods from the DrawingTool class (forward, turnLeft).

3. Remember from A2 that the general syntax of a method is:

```
modifiers return_type method_name ( parameters ){  
    method_body  
}
```

1. Parameters and arguments are used to pass information to a method. Parameters are used when defining a method, and arguments are used when calling the method.
2. In Code Sample 4-1, we use the `DrawingTool`'s `forward` method to move the `myPencil` object. However, we must tell the `forward` method how far to move, or it would not be a very useful method. We do this by passing an argument to it. In our example, we sent it the `int` value 100. The `turnLeft` method will default to 90 degrees if we don't pass it a value. If we want it to turn a different amount, we can send how far left we want it to turn in degrees.

```
myPencil.turnLeft(60);
```

3. When a method is called with an argument, the parameter receives the value of the argument. If the data type is primitive, we can change the value of the parameter within the method as much as we want and not change the value of the argument passed in the method call. Object variables, however, are references to the object's physical location. When we pass an object's variable name, we get a copy of that reference. Therefore, when we use the passed in reference to access this object within the method, we are in fact working with the same data of the object that was passed as an argument and have the ability to directly change the data inside the object. This can get very messy if the programmer doesn't realize what is going on.
4. When defining a method, the list of parameters can contain multiple parameters. For example:

```
public double doMath(int a, double x){  
    ... code ...  
    return doubleVal;  
}
```

When this method is called, the arguments fed to the `doMath` method must be the correct type and must be in the same order. The first argument must be an `int`. The second argument can be a `double` or an `int` (since an `int` will automatically be converted by Java).

```
double dbl = doMath(2, 3.5);      // this is okay  
double db2 = doMath(2, 3);        // this is okay  
double db3 = doMath(1.05, 6.37);  // this will not compile
```

5. Arguments can be either literal values (2, 3.5) or variables (a, x).

```
int a = 5;
int x = 10;

double db4 = doMath(5, 10); // example using literal
values
double db5 = doMath(a, x);  // example using variables
//These are equivalent calls
```

C. Returning Values

page 5 of 11

1. Sometimes we want a method to return a value.
2. In order for a method to return a value, there must be a return statement somewhere in the body of the method.
3. You must also declare which type of data value will be returned in the method declaration. This can be any primitive data type, a Java class, or any class that you have defined yourself.

```
public int getNumber(){
    int myNumber = 1234;
    return myNumber;
}
```

4. If a method returns no value, the keyword void should be used. For example:

```
public void printHello(){
    System.out.println("Hello world");
}
```

D. The Signature of a Method

page 6 of 11

1. In order to call a method legally, you need to know its name, how many parameters it has, the type of each parameter, and the order of those parameters. This information is called the method's *signature*. The signature of the method `doMath` can be expressed as:

```
doMath(int, double)
```

Note that the signature does not include the names of the parameters. If you just want to use the method, you don't need to know what the parameter names are, so the names are not part of the signature.

2. Java allows two different methods in the same class to have the same name, provided that their signatures are different. We say that the name of the method is *overloaded* because it has different meanings. The Java compiler doesn't get the methods mixed up. It can tell which one you want to call by the number and types of the arguments that you provide in the call statement. You have already seen overloading used in the `System.out` object, which is an instance of the `PrintStream` class. This class defines many different methods named `println`. These methods all have different signatures, such as:

```
println(int)           println(double)           println(String)
println(char)          println(boolean)          println()
```

In addition to these, we have been using this concept with the `DrawingTool` class and its `turnLeft` method.

```
turnLeft()             turnLeft(120)
```

3. It is illegal to have two methods in the same class that have the same signature but have different return types. For example, it would be a syntax error for a class to contain two methods defined as:

```
double doMath(int a, double x){}
int doMath(int a, double x){}
```

The Java compiler cannot differentiate return values so it uses the signature to decide which method to call.

E. Lifetime, Initialization, and Scope of Variables

page 7 of 11

1. Three categories of Java variables have been explained thus far in these lessons.
 - Instance variables
 - Local variables
 - Parameters

2. The lifetime of a variable defines the portion of runtime during which the variable exists. When an object is constructed, all its instance variables are created. As long as any part of the program can access the object, it stays alive. A local variable is created when the program executes the statement that defines it. It stays alive until the block that encloses the variable definition is exited. When a method is called, its parameters are created. They stay alive until the method returns to the caller.
3. The type of the variable determines its initial state. Instance variables are automatically initialized with a default value (0 for numbers, `false` for `boolean`, `null` for objects). Parameters are initialized with copies of the arguments. Local variables are not initialized by default so an initial value must be supplied. The compiler will generate an error if an attempt is made to use a local variable that may not have been initialized.
4. Scope refers to the area of a program in which an identifier is valid and has meaning. Instance variables are usually declared `private` and have class scope. Class scope begins at the opening left brace ({} of the class definition and terminates at the closing brace (}). Class scope enables methods to directly access all of its instance variables. The scope of a local variable extends from the point of its definition to the end of the enclosing block. The scope of a parameter is the entire body of its method.
5. An example of the scope of a variable is given in Code Sample 4-2. The class `ScopeTest` is created with three methods:

```

public class ScopeTest{
    private int test = 30;

    public void printLocalTest(){
        int test = 20;
        System.out.println("LocalTest:  " + test);
    }

    public void printClassTest(){
        System.out.println("ClassTest:  " + test);
    }

    public void printParamTest(int test){
        System.out.println("ParamTest:  " + test);
    }
}

//a driver to run the above class
public class ScopeDriver{
    public static void main (String[ ] args){
        int test = 10;

        ScopeTest st = new ScopeTest();
        System.out.println("main:  test = " + test);

        st.printLocalTest();
        st.printClassTest();
        st.printParamTest(40);
    }
}

```

Run output:

```

main: test = 10
LocalTest: 20
ClassTest: 30
ParamTest: 40

```

Code Sample 4-2

6. The results show the following about the scope of the variable test:

- Within the scope of main, the value of test is 10, the value assigned within the main method.
- Within the scope of printLocalTest, the value of test is 20, the value assigned within the printLocalTest method
- Within the scope of printClassTest, the value of test is 30, the private value assigned within ScopeTest, because there is no value given to test within the printClassTest method
- Within the scope of printParamTest, the value of test is 40, the value sent to the

1. When you are first starting to program, some of the most commonly used methods are called Getters and Setters (some like to call them Mutators and Accessors). These methods deal directly with attributes of the object they are associated with.
2. When we are working with an object, sometimes we will need to know certain pieces of information about the object that only the object can tell us reliably. Recall the `DrawSquare` class that we worked with earlier. We might remember the length of a side, but if that length has changed for some reason during the life of the object, we might be in for a surprise if we used that value. Here we would want to store the side length in a private instance variable and provide a Getter method along the lines of `double getSideLength()`. The Getter method's purpose would be to give us current information about that object. We don't want to let other objects access the side length directly, because they might alter that data when we don't want them to. The `getSideLength` method allows other objects to look at the data in our `DrawSquare` object and storing the length in a private variable prevents these objects from directly accessing the length.
3. What happens if we do want to change that side length from outside our class? As it stands right now, that would not be possible. However, we could create a Setter method in the `DrawSquare` class that would do this for us, `void setSideLength(double d)`. This method's basic purpose is to change the value of the sides for us, but it could also do a bit more. For instance, what if someone passed `setSideLength` a value of negative 10? Our square could obviously not exist with a negative side value, so our Setter program should check for validity of the values. Setters can often do calculations for us when necessary, so they are not always simply changing a value and doing nothing else. Often, Setter methods are given a return type of `boolean` which will return `true` if the value was valid and `false` if the value was not. This lets clients know if their value was accepted or not. If the client sends an invalid value to a method, it is usually good for them to know that they tried to use the method incorrectly.

As you become more experienced, your programs will grow in size and complexity. When this happens, it becomes more difficult to get your program to do what it is meant to do. Breaking a large task up into smaller tasks by using methods helps make the process easier. Designing methods is complex; the design must integrate a parameter list, the return value, and the goal of the method. Methods should accomplish small tasks within your classes in an easy and organized way. When your programs start to become very large, good commenting and formatting habits will help you stay organized. Knowing these tools will also help you read code written by other people, which is a great way to learn.

Chapter 5

A5 Introduction

page 1 of 11

This lesson discusses how to design your own classes. This can be the most challenging part of programming. A truly good design can be the difference between hundreds of hours working with complex code and two hours working in an elegant system. A well thought out design can make the programming portion far easier. In fact, for many professional projects, more time is spent designing programs than actually typing in code. Imagine a million lines of code in a project with a design flaw. Redesigning that much code could be horrendous!

The key topics for this lesson are:

- A. [Designing a Class](#)
- B. [Determining Object Behavior](#)
- C. [Instance Variables](#)
- D. [Implementing Methods](#)
- E. [Constructors](#)
- F. [Using Classes](#)

[A5 Vocabulary](#)

page 2 of 11

ACCESS SPECIFIER
BEHAVIORS
ENCAPSULATION
OVERLOADING
REFERENCE
VARIABLE

ATTRIBUTES
CONSTRUCTOR
INSTANCE VARIABLE
PSEUDOCODE
TOP DOWN DESIGN

A. Designing a Class

page 3 of 11

1. One of the advantages of object-oriented design is that it allows a programmer to create a new data type that is reusable in other situations.
2. When designing a new class, three components must be identified — attributes, behaviors, and constructors. To determine attributes of a class, look at the nouns associated with that object. To determine behaviors, look at the verbs.
3. Let's consider a checking account at a bank. The account would need to record such things as the account number, the current balance, the type of checking account it is, etc (these are nouns). These would be the attributes of the checking account. It would also need to be able to do certain actions, such as withdrawing or depositing money (these are verbs). These would be the behaviors of the checking account. Finally, the checking account object needs to be created in order to be used, so the class must define how the creation process works. This is accomplished in the constructors.

B. Determining Object Behavior

page 4 of 11

1. In this section, you will learn how to create a simple class that describes the behavior of a bank account. Before you start programming, you need to understand how the objects of your class behave. Operations that can be carried out with a checking account could be:
 - Accepting a deposit
 - Withdrawing from the account

- Getting the current balance

2. In Java, these operations are expressed as *method calls*. For example, assume we have an object `checking` of type `CheckingAccount`. Here are the methods that invoke the required behaviors:

```
checking.deposit(1000);  
checking.withdraw(250);  
System.out.println("Balance: " + checking.getBalance());
```

These methods form the behaviors of the `CheckingAccount` class. The behaviors are the list of methods that you can apply to objects of a given class. To the client, an object of type `CheckingAccount` can be viewed as a “black box” that can carry out its methods. The programming concept of not needing to know how things are done internally is called abstraction.

3. Once we understand what objects of the `CheckingAccount` class need to do, it is possible to design a Java class that implements these behaviors. To describe object behavior, you first need to implement a class and then implement methods within that class.

```
public class CheckingAccount{  
    // CheckingAccount data  
  
    // CheckingAccount constructors  
  
    // CheckingAccount methods  
}
```

Next we implement the three methods that have already been identified:

- `deposit`
- `withdraw`
- `getBalance`

```

public class CheckingAccount{
    // CheckingAccount data

    // CheckingAccount constructors

    public void deposit( double amount ){
        // method implementation
    }
    public void withdraw( double amount ){
        // method implementation
    }

    public double getBalance(){
        // method implementation
    }
}

```

4. What we have been doing here is not real code and wouldn't actually do anything. However, it is useful to lay out what your class will look like. When we use a mixture of English and Java to show what we are doing, it is called *pseudocode*. In this example the implementation of the methods is left out because we do not have all the information that we need yet. However, we can still write out what the methods will do with pseudocode so that it becomes easier to see how everything will fit together. This process of starting with a very broad concept or outline and working down to smaller and smaller details is called *top-down* design.

```

public class CheckingAccount{
    // CheckingAccount data

    // CheckingAccount constructors

    public void deposit( double amount ){
        // receive the amount of the deposit
        // and add it to the current balance
    }
    public void withdraw( double amount ){
        // remove the amount of the withdrawal
        // from the current balance
    }

    public double getBalance(){
        // return the current balance in a double value
    }
}

```

5. A method header consists of the following parts:

access_specifier return_type method_name (parameters)

- a. An ***access_specifier*** (such as **public**). The access specifier controls

where this method can be accessed from. Methods should be declared as public if the method needs to be accessed by something other than the object containing the method. If it should only be accessed within the object, you should declare the method as private.

- b. The *return_type* of the method such as **double**, **void**, or **DrawingTool**. The return type is the data type that the method sends back to the call of the method. This can be any primitive type or any object that your class knows about. For example, in the `CheckingAccount` class, the `getBalance` method returns the current account balance, which is a floating-point number, so its return type is **double**. The `deposit` and `withdraw` methods don't return any value. To indicate that a method does not return a value, you use the keyword **void**.
 - c. The *method_name* (such as `deposit`). The name needs to follow the rules of identifiers and should indicate the method's purpose.
 - d. A list of the *parameters* of the method. The parameters are the input to the method. The `deposit` and `withdraw` methods each have one parameter, the amount of money to deposit or withdraw. The type of parameter, such as **double**, and name for each parameter, such as `amount`, must be specified. If a method has no parameters, like `getBalance`, it is still necessary to supply a pair of parentheses `()` behind the method name.
6. Once the method header has been specified, the implementation of the method must be supplied in a block that is delimited by braces `{...}`. The `CheckingAccount` methods will be implemented later in Section D.

C. Instance Variables

page 5 of 11

1. Before any code can be written for the behaviors, the object must know how to store its current *state*. The state is the set of attributes that describes the object and that influences how an object reacts to method calls. In the case of our checking account objects, the state includes the current balance and an account identifier.
2. Each object stores its state in one or more *instance variables*.

```

public class CheckingAccount{
    private double myBalance;
    private int myAccountNumber;

    // CheckingAccount constructors

    // CheckingAccount methods
}

```

3. An instance variable declaration consists of the following parts:

access_specifier type variable_name

- a. The ***access_specifier*** (such as **private**) tells who can access that data member. Instance variables are generally declared with the access specifier **private**. That means they can be accessed only by methods of the same class. In particular, the balance variable can be accessed only by the deposit, withdraw, and getBalance methods.
 - b. The ***type*** of the variable (such as **double**).
 - c. The ***variable_name*** (such as myBalance).
4. If instance variables are declared private, then all external data access must occur through the non-private methods. This means that the instance variables of an object are hidden. The process of hiding data is called encapsulation. Although it is possible in Java to define instance variables as public (leave them unencapsulated), it is very uncommon in practice. In this curriculum, instance variables will always be made private.
5. For example, because the myBalance instance variable is private, it cannot be accessed from outside of the class:

```
double balance = checking.myBalance; // compiler ERROR!
```

However, the public getBalance method to inquire about the balance can be called:

```
double balance = checking.getBalance(); // OK
```

1. Now that we know how the object stores its state, we can provide the implementations for the methods of the class. The implementation for three methods of the CheckingAccount class is given below.

```
public class CheckingAccount{
    private double myBalance;
    private int myAccountNumber;

    public double getBalance(){
        return myBalance;
    }

    public void deposit(double amount){
        myBalance += amount;
    }

    public void withdraw(double amount){
        myBalance -= amount;
    }
}
```

2. The implementation of the methods is straightforward. When some amount of money is deposited or withdrawn, the balance increases or decreases by that amount.
3. The getBalance method simply *returns* the current balance. A **return** statement obtains the value of a variable and exits the method immediately. The return value becomes the value of the method call expression. The syntax of a **return** statement is:

return expression;

or

return; // Exits the method without sending back a value

E. Constructors

page 7 of 11

1. The final requirement to implement the checkingAccount class is to define a *constructor*, whose purpose is to initialize the values of instance variables of an object. To construct objects of the CheckingAccount class, it is necessary to declare an object variable first.

```
CheckingAccount checking;
```


Object variables such as `checking` are *references* to objects. Instead of holding an object itself, a reference variable holds the information necessary to find the object in memory. This is the *address* of the object.

2. The object identifier `checking` does not refer to any object yet. An attempt to invoke a method on this variable would cause a runtime null pointer exception error. To initialize the variable, it is necessary to create a new `CheckingAccount` object using the `new` operator

```
checking = new CheckingAccount();
```

Constructors are always invoked using the `new` operator. The `new` operator allocates memory for the objects, and the constructor initializes it. The “new” operator returns the reference to the newly constructed object.

In most cases, you will declare and store a reference to an object in an object identifier on one line as follows:

```
CheckingAccount checking = new CheckingAccount();
```

Occasionally, it would be repetitive and unnecessary to create an object identifier. If the purpose of creating the object is only to pass it in as an argument, you can simply create the object within the method call. For example, when creating `DrawingTool` objects, and you are providing a `SketchPad` object, you do not need to create an identifier for that `SketchPad` object:

```
DrawingTool pen = new DrawingTool(new SketchPad(500,500));
```

Notice that we never create an object identifier for the `SketchPad` object.

3. Constructors always have the same name as their class. Similar to methods, constructors are generally declared as `public` to enable any code in a program to construct new objects of the class. Unlike methods, constructors do not have return types.
4. Instance variables are automatically initialized with a default value (0 for number, false for boolean, null for objects). Even though initialization is handled automatically for instance variables, it is a matter of good style to initialize all instance variables explicitly. Generally, all of your instance variables should be initialized in your constructor.
5. Many classes define more than one constructor through overloading. For example, you can supply a second constructor for the `CheckingAccount` class that sets the `myBalance` and `myAccountNumber` instance variables to initial values, which are the parameters of the constructor:

```

public class CheckingAccount{
// CheckingAccount data
    private double myBalance;
    private int myAccountNumber;

// constructor initializes values to default settings
    public CheckingAccount(){
        myBalance = 0.0;
        myAccountNumber = 0;
    }

    public CheckingAccount(double initialBalance, int acctNum){
        myBalance = initialBalance;
        myAccountNumber = acctNum;
    }
// CheckingAccount methods
}

```

The second constructor is used if you supply a starting balance and an account number as construction parameters.

```
CheckingAccount checking = new CheckingAccount(5000.0, 12345);
```

The number of constructors is based on the needs of the client.

6. The implementation of the CheckingAccount class is complete and given below:

```

public class CheckingAccount{
    private double myBalance;
    private int myAccountNumber;

    public CheckingAccount(){
        myBalance = 0.0;
        myAccountNumber = 0;
    }

    public CheckingAccount(double initialBalance, int acctNum){
        myBalance = initialBalance;
        myAccountNumber = acctNum;
    }

    public double getBalance(){
        return myBalance;
    }

    public void deposit(double amount){
        myBalance += amount;
    }

    public void withdraw( double amount ){
        myBalance -= amount;
    }
}

```

F. Using Classes

page 8 of 11

1. Using the `CheckingAccount` class is best demonstrated by writing a program that solves a specific problem. We want to study the following scenario:

An interest bearing checking account is created with a balance of \$1,000. For two years in a row, add 2.5% interest. How much money is in the account after two years?

2. Two classes are required: the `CheckingAccount` class that was developed in the preceding sections, and a second class called `checkingTester`. The main method of the `checkingTester` class constructs a `CheckingAccount` object, adds the interest twice, then prints out the balance.

```

class CheckingTester{
    public static void main(String[] args){
        CheckingAccount checking =
            new CheckingAccount(1000.0, 123);

        double INTEREST_RATE = 2.5;
        double interest;

        interest = checking.getBalance() * INTEREST_RATE / 100;
        checking.deposit(interest);

        System.out.println("Balance after year 1 is $"
            + checking.getBalance());

        interest = checking.getBalance() * INTEREST_RATE / 100;
        checking.deposit(interest);

        System.out.println("Balance after year 2 is $"
            + checking.getBalance());
    }
}

```

The topics in this lesson are critical to your study of computer science. The concepts of abstraction and OOP will continue to be developed in future lessons. Designing your classes is the most important part of programming. Without good design in the beginning, a complex program can quickly grow out of control.

Chapter 6

Now that you have learned how to design your own classes, we will explore how to take advantage of the huge number of pre-made classes provided with Java. We will also learn

how to read the APIs that come with those classes so that you will be able to take any class that comes with an API and teach yourself how to use that class. In this chapter, we will start with APIs, explore a few useful classes and their APIs, and then finish by learning how to write our own APIs so that other people can use our classes.

The key topics for this lesson are:

- A. [Understanding APIs](#)
- B. [Final and Static](#)
- C. [DrawingTool](#)
- D. [Point2D.Double](#)
- E. [Random](#)
- F. [Math](#)
- G. [Javadoc Tool](#)

[A6 Vocabulary](#)

page 2 of 12

API	PACKAGES
STATIC	FINAL
JAVADOC	

A. Understanding APIs

page 3 of 12

1. API stands for Application Programming Interface and is one of the most useful tools you will have while working with Java. APIs show exactly how to use pre-made classes. The DrawingTool Class Specifications handout in Lesson A1 is an example of a simplified API. It lists the classes and constructors so that we know which methods are available. APIs do not tell us how the programmer dealt with a problem or what kind of formulas they used internally, but just tells us what methods we can access, how to interact with those methods, and what those methods will return back to us.
2. You can always access the Java APIs at java.sun.com. Click on API Specifications on the main page and then choose the version of Java you wish to retrieve the API for.

You can also download the APIs to your computer for offline access. Many Java programming environments can be set up to access the APIs from within your code.

3. The Java APIs are organized both by package and by class. Packages are groups of related classes that are “packaged” together. When you use the code `import gpdraw.*;` you are adding the entire gpdraw package to your code. If you only need one or two classes from a package, you can add the classes individually with the code `import gpdraw.DrawingTool;`

B. Final and Static

page 4 of 12

1. Before you look at these APIs in depth, you need to learn what the keywords *final* and *static* mean, as they frequently come up in the API documentation.
2. When used with a primitive data type, final means that the value of the variable will never change. This is useful in many cases, such as tax rates, math constants such as PI, and base values that are used in several places in your code. Identifiers with final are generally made with only capital letters so they are easily distinguishable from the rest of the code.

```
final double TAXRATE = 0.0825;  
final double ROUNDS_IN_GAME = 100;
```

3. A program that repeatedly uses a constant identifier, such as TAXRATE, can be quickly modified by assigning a value to TAXRATE at the top of the program. If the value of the tax rate is hard coded everywhere it is used, then changing the tax rate would involve changing that value everywhere it appears in the program. This involves searching through your program and finding every single time that value is used, and then changing it to the new value. It would be easy to accidentally miss a value or two or possibly change something that was similar but wasn't supposed to be the tax rate. Using final values will not only save time but can also reduce the number of errors in your program.

```
import gpdraw.*;
```

4. Once a final variable is given a value within a program, that value may never change in that run. In order to change that value, you must change it within your code, recompile, and run the program again. You may still define this value within the constructors so that the value can be determined at run time; it need not be defined at the same time that the variable is declared.

5. Objects and methods can also take a final keyword. However, they behave differently than primitive data types. We have not yet discussed the concepts that these final objects and methods affect, but we will cover them in Lesson A11 - Inheritance.
6. Using the keyword static means that the data member or method is attached to the class rather than to an object of that class.
7. With a static method, you only need to type the name of the class followed by the name of the method. You never need to create a new object when dealing with static methods. You can think of static methods as belonging to the class itself, whereas non-static methods are attached to the objects created from that class.

```
int jason = Math.pow(3,4);
```

jason receives and stores the result of 3 to the 4th power, which is 81.

Notice how there was never any need to create an object of type Math. Instead, we just have an int assigned the value created by the static pow method of the Math class.

8. Data members that are static may also be used without creating an object of that class. They also have specific behavior when using that value of the data member within the class. While we may make a hundred objects from one class, any static variables of that class will in fact be shared by each of those objects. If one object changes the value of that static variable, then the value is changed for all of the other objects of that type. This is because the variable and its value do not belong to any of the objects individually, but to the class itself.

C. DrawingTool

page 5 of 12

1. You have already been looking at an API for DrawingTool. The purpose of Handout A1.1 was to give you an introduction to the purpose of the class and how to use its various methods.
2. When instructed to draw a circle, you probably looked at Handout A1.1 and saw this:

```
public drawCircle (double r);
```

postcondition

- If the object is in drawing mode, a circle of radius `r` is drawn around the current location using the current `width` and `color`.

This tells us exactly what we need to know to use this method. We have the name of the method and the type of argument it takes. We also know what will happen after the method is called.

3. This is not the official Java format for an API, but it accomplishes the same thing. Without this handout, how would you have known how to draw a circle? How about when you made the picture of the house? How would you have known to use the `forward` method, `turnLeft`, `down`, or `up`? As you can see, APIs are an essential tool that must be looked at before a programmer can understand how to use pre-made classes.
4. Now that you understand how to read the `DrawingTool` API, take a look at a sample (the Pizza Parlor assignment) in the Javadoc folder for this lesson. Open up the `index-all.html` file in your Web browser to see a basic package view. Click on the Help Link at the top to access a page on “How This API Document is Organized”. Links that do not work are simply placeholders. The feature that you are clicking on does not exist for that class.

D. `Point2D.Double`

page 6 of 12

1. The `Point2D` class is useful for storing locations on a two dimensional space. It also contains several methods that can be used to do certain calculations. There are two sub-classes of `Point2D`, but you will generally only want to use `Point2D.Double`.
2. Consider an application where you need to track the locations of two mice in a flat-bottomed box. You could use two `Point2D.Double` objects to keep track of their locations.
3. What if we want to be able to tell how far apart the two mice are at any given time? We could use the distance formula from Geometry to calculate the distance. However, if we take a quick look at the `Point2D` API, we can find this method:

<code>double</code>	<code>distance(Point2D pt)</code> Returns the distance from this <code>Point2D</code> to a specified <code>Point2D</code> .
---------------------	---

A line of code as simple as

```
double distance = rat1.distance(rat2);
```

will give us the distance between the two rats. This is much simpler than trying to do all the calculations ourselves. Remember, whenever possible we want to avoid writing code that has already been written. By doing a slight bit of research, we have saved ourselves the hassle of writing and then debugging code.

E. Random

page 7 of 12

1. As the name suggests, the `java.util` package is full of utility classes that you will find very useful. Many of them are going to be far too advanced for what you need at this stage, but as you progress in skill you should browse through the classes and see which ones you begin to understand. During this course, you will learn several of the `util` classes in detail, but for now we will just concentrate on this one, `java.util.Random`.
2. `Random` is probably the most fun of all the classes in the `java.util` package. Any sort of game or in depth simulation (such as inspecting a transportation system to see how efficient it is) will generally need some sort of randomness in order to work the way we want. That's where the `Random` class comes in. Let's take a look at it now.

Constructor Summary

Random() Creates a new random number generator.	
---	--

Method Summary

double	nextDouble() Returns the next pseudorandom, uniformly distributed double value between 0.0 and 1.0 from this random number generator's sequence.
int	nextInt(int n) Returns a pseudorandom, uniformly distributed int value between 0 (inclusive) and the specified value (exclusive), drawn from this random number generator's sequence.

3. These are the methods that you will be likely to find the most useful. The constructor is very basic and requires no arguments. The word “pseudorandom” just indicates that the number is not completely random. Computers are not physically capable of creating true random numbers. Therefore, in Computer Science, we refer to computer generated random numbers as “pseudorandom.”
4. Let’s look at a situation where we might use this class. Consider an electronic raffle for a prize. There are 200 participants with one number each between 1 and 200. We can create an object of type Random to determine who the winner is.

```
Random chooser = new Random();  
int winner = chooser.nextInt(200) + 1;
```

This code will give us a value between 1 and 200 in our winner variable.

5. The Random class has many uses. Most board games you have played probably use six sided dice to give the game an element of chance. Video and computer games also use randomness to determine whether you hit your enemy or not, modified by the skill of the character you are using. Think about the lottery as well. Without an element of chance, it would be pretty boring to buy lottery tickets and the game would cease to exist.

F. Math

page 8 of 12

1. The Math class in the `java.lang` package contains class methods for commonly used mathematical functions. Java loads the `java.lang` package automatically, so no special actions are required to access these.
2. The Math class contains both methods and the numerical values for two important mathematical constants, `e` and `Pi`. These constant values are accessed the same way as normal variables, but they can never be modified directly by your code.
3. The Math class is most useful for complex mathematical formulas, for example:

$$\frac{1}{2} \sin\left(x - \frac{\pi}{y^3}\right)$$

Using the Math class, we can create a line of code to solve this calculation much like you would type the same equation into your calculator:

```
(1.0/2.0) * Math.sin(x - Math.PI / Math.pow(y, 3));
```

4. Let's see what other kind of methods the Math class provides for us. Go ahead and take a look at the Java APIs. (Remember: Access the Java APIs on the Web at java.sun.com - click on API Specifications in the left column, and then pick the version of Java, such as [J2SE 1.4.2](#), that you wish to see.) Find the Math class within the Java.lang package. Hint: You can find it either in the long list of classes on the left hand side by scrolling down to the M section, or you can access the Java.lang package first in the upper left section and then look for the Math class. You should find something similar to this:

java.lang

Class Math

[java.lang.Object](#)

└─ [java.lang.Math](#)

```
public final class Math
extends Object
```

Field Summary

static double	E The double value that is closer than any other to e, the base of the natural logarithms.
static double	PI The double value that is closer than any other to pi, the ratio of the circumference of a circle to its diameter.

Method Summary

static double	abs(double a) Returns the absolute value of a double value.
static int	abs(int a) Returns the absolute value of an int value.
static double	pow(double a, double b) Returns the value of the first argument raised to the power of the second argument.

static double	sqrt (double a) Returns the correctly rounded positive square root of a double value.
------------------	---

- The list shown here is much shorter than you will find online. However, let's look at the layout. At the very top is `java.lang`, the name of the package this class is in. Below that is the name of the class, followed by a series of class names. You will learn more about this later, but for now just think of the classes listed there as the parents of the current class. Below that comes the description of the class, which gives an introduction to the purpose of the class. This allows programmers to quickly decide if this class will do what they need. Next is the list of attributes and behaviors, labeled as "Field Summary" and "Method Summary."
- The "Field Summary" section contains two items in it. In the left side of the table, we can see they are both labeled as `static double`. This tells us the type of the variable so we know how to use it. On the right, we see the name of the variables followed by a brief description of it. If you are looking at this on the Internet, you can click on the name label (E or PI) to be taken to a more detailed description of the variable. We don't generally need more information on variables, but the links are provided in the API just in case.
- The "Method Summary" section has all of the available methods provided by the `Math` class. Once again, we can see that the table is laid out in a similar manner, with a small section on the left and a larger section on the right. With the first listed method, `abs`, we can see the left section has the same label (`static double`) as we found with the E and PI values. However, this time it is telling us the return type of the methods. On the right, we again get the name and a brief description but we also get the arguments that must be passed to the method. Clicking on the name to go to the full description of methods is often much more useful than with the variables. If we click on the name `abs` we find much more information than we originally had with the short description:

abs	<pre>public static double abs(double a)</pre> <p>Returns the absolute value of a double value. If the argument is not negative, the argument is returned. If the argument is negative, the negation of the argument is returned. Special cases:</p> <ul style="list-style-type: none"> If the argument is positive zero or negative zero, the result is positive zero.
------------	--

- If the argument is infinite, the result is positive infinity.
- If the argument is NaN, the result is NaN.

In other words, the result is the same as the value of the expression:
`Double.longBitsToDouble((Double.doubleToLongBits(a)<<1)>>>1)`

Parameters:

a - the argument whose absolute value is to be determined

Returns:

the absolute value of the argument.

8. Here we gain much more information about the abs method and what its purpose is. We learn some basics about what it does to the value. It even gives us some special cases that, while rare, can still occur. Note: NaN means Not a Number.
9. Here are some examples using the abs method.

`Math.abs(25) -> 25` `Math.abs(-25) -> 25`

`Math.abs(0) -> 0` `Math.abs(17/5) -> 3`

G. Javadoc Tool

page 9 of 12

1. The basics of creating your own APIs are pretty simple. When you add comments in your code, you can use the tag `/**...*/` before each class, variable, constructor, and method to create block comments. These comments work within your code in essentially the same way as the regular block comment tag `/*...*/`. However, once we run the Javadoc tool, APIs will be created based on these comments.
2. The first line of the comment should be a quick description that sums up what it is in front of. This first line will turn into the quick description that we discussed earlier. The rest of your paragraph should consist of a more detailed description of the item.
3. When you run your javadoc.exe program on your Java class file (as discussed in [Handout A6.1](#), Javadocs), it will create a few .html files in the local directory. If you open up index.html with your Web browser, you will find yourself looking at an API created for your class.

As a Java programmer, you want to avoid redoing work already done. This is why we try to design methods and classes that do small jobs and can therefore be reused in other spots. Pre-made java classes help with this process by providing a huge library of commonly needed classes. APIs are our instruction books, which we use to understand the purpose and applicability of these classes.

Chapter 7

A7 Introduction

page 1 of 7

The input and output of a program's data is usually referred to as I/O. There are many different ways that a Java program can perform I/O. In this lesson, we present some very simple ways to handle text input typed in at the keyboard as well as how to format text to the screen. The Advanced Placement subset does not require that you know how to use any specific input and output classes, only that you know how to use I/O in some manner. This curriculum will use the Scanner class and the `printf()` method provided with Java 1.5.

The key topics for this lesson are:

- A. [Reading Input with the Scanner Class](#)
- B. [Multiple Line Stream Output Expressions](#)
- C. [Formatting Output](#)

[A7 Vocabulary](#)

page 2 of 7

CONVERSION

FLAGS

Created by mu

PRECISION
Scanner
System.out

printf
System.in
WIDTH

A. Reading Input with the Scanner Class

page 3 of 7

1. Some of the programs from the preceding lessons have been written without any flexibility. To change any of the data values in the programs, it is necessary to change the variable initializations, recompile the program, and run it again. Sometimes prompting the user for a value and then processing the data is more efficient and convenient.
2. However, accepting user input in Java can be complex. Throughout this curriculum, we will use the `Scanner` class to make processing input easier and less tedious.
3. Just as the `System` class provides `System.out` for output, there is an object for input, `System.in`. Unfortunately, Java's `System.in` object does not directly support convenient methods for reading numbers and strings. We need to have a class sitting between the `System.in` object and ourselves to filter what comes through. This is what the `Scanner` class does.
4. `Scanner` is part of the `java.util` package, so we need to start off by adding the `Scanner` class to our import section.

```
import java.util.Scanner;
```

5. Next, we create our `Scanner` object and pass in the `System.in` object.

```
Scanner in = new Scanner(System.in);
```

This tells the `Scanner` to look at the `System.in` object for all of its input. In Student Lesson A13, we will learn how to change the object we pass to the `Scanner` so that we can read the data stored in text files.

6. Here are some example statements:

```

int num1;
double bigNum;
boolean isTrue;

num1 = in.nextInt( );
bigNum = in.nextDouble( );
isTrue = in.nextBoolean( );

```

When the statement `num1 = in.nextInt()` is encountered, the program pauses until an appropriate value is entered on the keyboard.

7. Any whitespace (spaces, tabs, newline) will separate input values. When reading values, whitespace keystrokes are ignored.
8. When requesting data from the user via the keyboard, it is good programming practice to provide a prompt. An unIntroduced input statement leaves the user hanging without a clue of what the program wants. For example:

```

System.out.print("Enter an integer --> ");
number = in.nextInt();

```

B. Multiple Line Stream Output Expressions

page 4 of 7

1. We have already used examples of multiple output statements such as:

```

System.out.println("The value of sum = " + sum);

```

2. When the length of an output statement exceeds one line of code, it can be broken up several different ways:

```

System.out.println("The sum of " + num1 + " and " + num2 +
    " = " + (num1 + num2));

```

or

```

System.out.print("The sum of " + num1 + " and " + num2);
System.out.println( " = " + (num1 + num2));

```

3. You cannot break up a String constant and wrap it around a line.

```

System.out.print("A long string constant must be broken
up into two separate quotes. This will NOT work.");

```



```
System.out.print("A long string constant must be broken up"
+ " into two separate quotes. This will work.");
```

C. Formatting Output

page 5 of 7

1. To format, we will learn a new printing method, `printf()`. It works similarly to the `print()` and `println()` methods that you have already been using to output text.
2. The `printf()` method takes two arguments. The first one is the formatting String, a special sequence of characters that tells `printf()` how to display the second argument. The syntax for the formatting String is:

`%[flags][width][.precision]conversion`

The '%' sign tells the `printf` method that formatting is coming. All of your formatted String constants will start with %. It does not have to be the very first thing in your String constant, just the first part of any formatted text.

3. The last part of the formatting String, conversion, is one of the most important parts. It is what determines how the `printf()` method reacts to a message you send it. The most important conversion tags for you to know are 's', 'd', and 'f'. 'd' is used for integers (base-10 notation), 'f' is for numbers with decimal places (doubles), and 's' is for String literals. The conversion tag always comes at the end of the formatting String.

Conversion Tag	Usage Type	Example
s	String literals	<code>printf("%s", "Sam")</code>
d	ints	<code>printf("%d", 5182)</code>
f	doubles	<code>printf("%f", 2.123456)</code>

4. Precision is very easy and straightforward. When using a formatting String with the 's' conversion tag, this will tell `printf()` the maximum number of characters to print out. When used with the 'f' conversion tag, you can specify how many decimal places to print out, rounded to the closest number. If you don't specify how many decimal places to display with 'f' then it will default to six places.

```
System.out.printf("%.2s", "Hello") -> He
System.out.printf("%.10s", "Hello") -> Hello
System.out.printf("%.5f", Math.PI) -> 3.14159
System.out.printf("%.4f", Math.PI) -> 3.1416
System.out.printf("%f", Math.PI) -> 3.141593
```

5. Width tells `printf()` the minimum number of characters to print out. This allows for creating right-aligned lists or menus. `printf()` does not distinguish between normal characters and special characters (escape sequences), so "Prices:" and "Prices:\n" are two different sizes. If you want to print your data out left-aligned, you can simply add a '-' character to the left of the width value.

Note: In the example below, the first line has a width of 11 instead of 10 to adjust for the \n error. Because numeric entries cannot use these special characters, it is better to use `println()` to separate lines when utilizing `printf()`. The example below shows one instance of using the \n character as well as doing two columns of formatted output.

```
System.out.printf("%-10s", "Name:");
System.out.printf("%11s", "Price:\n");
System.out.printf("%-10s", "Soda");
System.out.printf("%10.2f", 10.25);
System.out.println();
System.out.printf("%-10s", "Candy");
System.out.printf("%10.2f", 1.50);
```

Output:

Name:	Price:
Soda	10.25
Candy	1.50

6. Flags are special characters that give special properties to the values passed in. Adding a '+' sign in the formatting String will give numbers a positive or negative sign when printed. Putting in a '(' will cause negative numbers to be enclosed with parentheses. The most useful of the flags is ',' because it will add commas into large numbers (in the correct spot for the region, i.e. Japan puts numbers into groups of four unlike the US, which puts numbers in groups of three). To get a dollar sign directly before your printed value, place the '\$' character directly before the '%' sign.

```
System.out.printf("%,d", 12345678) -> 12,345,678
System.out.printf("$%,d", 12345678) -> $12,345,678
System.out.printf("%,(d", 12345678) -> (12,345,678
System.out.printf("%,(d", -12345678) -> (-12,345,678)
```

7. You may put multiple arguments in one call to `printf()` for organizational purposes. Simply put multiple formatting Strings in the first passed argument to `printf()` and then add your additional arguments, separated by commas.

```
double mySum = 123.456
System.out.printf("%10s %10.2f", "Total:", mySum);
```

8. This curriculum uses just a few of the many things that `printf()` is capable of. Once you get more familiar with using the formatting options shown in this guide, you can look at the API for the `Formatter` class for more information on formatting Strings. The `printf()` method is an easy way to manage the `Formatter` class in a way that is very similar to the common `print()` and `println()` functions.

Summary/Review

page 6 of 7

These two classes, `Scanner` and `Formatter`, will be used in many programs. They provide you with the flexibility needed to start creating very robust programs. You can now interact with a user instead of simply displaying results on the screen. Go back through some of the programs you have already made and see which ones could benefit from accepting user input or formatting data.

Chapter 8

A8 Introduction

page 1 of 17

Any sort of complex program must have some ability to control flow. Without this control, programs become limited to one basic job each time the program is run. The most basic of these control structures is the **if** statement, followed by the **if-else**, and then the **switch** statement.

The key topics for this lesson are:

- A. [Structured Programming](#)
- B. [Control Structures](#)
- C. [Algorithm Development and Pseudocode](#)

- D. [Relational Operators](#)
- E. [Logical Operators](#)
- F. [Precedence and Associativity of Operators](#)
- G. [The if-else Statements](#)
- H. [Compound Statements](#)
- I. [Nested if-else Statements](#)
- J. [Conditional Operator](#)
- K. [Boolean Identifiers](#)
- L. [Switch Statements \(Optional\)](#)

[A8 Vocabulary](#)

page 2 of 17

ALGORITHM	BOOLEAN IDENTIFIER
COMPOUND STATEMENT	CONDITIONAL OPERATOR
CONTROL STRUCTURE	IF-ELSE
ITERATION	LOGICAL OPERATOR
PSEUDOCODE	RELATIONAL OPERATOR
STEPWISE REFINEMENT	STRUCTURED PROGRAMMING

A. Structured Programming

page 3 of 17

1. In the early days of programming (1960's), the approach to writing software was relatively primitive and ineffective. Much of the code was written with **goto** statements that transferred program control to another line in the code. Tracing this type of code was an exercise in jumping from one spot to another, leaving behind a trail of lines similar to spaghetti. The term "spaghetti code" comes from trying to trace code linked together with **goto** statements. The complexity this added to code led to the development of structured programming.
2. The research of Bohm and Jacopini has led to the rules of structured programming. Here are five tenets of structured programming.
 - a. No **goto** statements are to be used in writing code.
 - b. All programs can be written in terms of three control structures: sequence, selection, and iteration.
 - c. Each control structure has one entrance point and one exit point. We will

sometimes allow for multiple exit points from a control structure using the **break** statement.

- d. Control structures may be stacked (sequenced) one after the other.
- e. Control structures may be nested inside other control structures.

3. The control structures of Java encourage structured programming. Staying within the guidelines of structured programming has led to great productivity gains in the field of software engineering.

B. Control Structures

page 4 of 17

1. There are only three necessary control structures needed to write programs: sequence, selection, and iteration.
2. Sequence refers to the line-by-line execution as used in your programs so far. The program enters the sequence, does each step, and exits the sequence. This allows for sequences to do only a limited job during each execution.
3. Selection is the control structure that allows choice among different paths. Java provides different levels of selection:
 - One-way selection with an **if** structure
 - Two-way selection with an **if-else** structure
 - Multiple selection with a **switch** structure
4. Iteration refers to looping. Java provides three loop structures. These will be discussed in length in Student Lesson A12.
 - **while** loops
 - **do-while** loops
 - **for** loops

1. An algorithm is a solution to a problem. Computer scientists are in the problem-solving business. They use techniques of structured programming to develop solutions to problems. Algorithms will range from the easier "finding the average of two numbers" to the more difficult "visiting all the subdirectories on a hard disk, searching for a file."
2. A major task of the implementation stage is the conversion of rough designs into refined algorithms that can then be coded in the implementation language of choice.
3. Pseudocode refers to a rough-draft outline of an answer, written in English-like terms. These generally use phrases and words that are close to programming languages, but avoid using any specific language syntax. Once the pseudocode has been developed, translation into code occurs more easily than if we had skipped this pseudocode stage.
4. Stepwise refinement is the process of gradually developing a more detailed description of an algorithm. Problem solving in computer science involves overall development of the sections of a program, expanding each section with more detail, later working out the individual steps of an algorithm using pseudocode, and then finally writing a code solution.

D. Relational Operators

1. A relational operator is a binary operator that compares two values. The following symbols are used in Java as relational operators:

<	less than
>	greater than
<=	less than or equal to
>=	greater than or equal to
==	equal to
!=	not equal to

2. A relational operator is used to compare two values, resulting in a relational expression. For example:

`number > 16`

`grade == 'F'`

`passing >= 60`

3. The result of a relational expression is a **boolean** value of either **true** or **false**.
4. When character data is compared, the ASCII code values are used to determine the answer. The following expressions result in the answers given:

'A' < 'B'	evaluates as true, (65 < 66)
'd' < 'a'	evaluates as false, (100 < 97)
't' < 'X'	evaluates as false, (116 < 88)

In the last example, you must remember that upper case letters come first in the ASCII collating sequence; the lower case letters follow after and consequently have larger ASCII values than do upper case ('A' = 65, 'a' = 97).

E. Logical Operators

page 7 of 17

1. The three logical operators in the AP subset are AND, OR, and NOT. These operators are represented by the following symbols in Java:

AND	&&
OR	(two vertical bars)
NOT	!

These logical operators allow us to combine conditions. For example, if a dog is gray and weighs less than 15 pounds it is the perfect lap dog.

2. The && (and) operator requires both operands (values) to be true for the result to be true.

```
(true && true) -> true
(true && false) -> false
(false && true) -> false
(false && false) -> false
```

3. The following are Java examples of using the && (and) operator.

```
((2 < 3) && (3.5 > 3.0)) -> true
((1 == 0) && (2 != 3)) -> false
```

The && operator performs short-circuit evaluation in Java. If the first operand in && statement is false, the operator immediately returns false without evaluating the second half.

4. The || (or) operator requires only one operand (value) to be true for the result to be true.

```
(true || true) -> true
(true || false) -> true
(false || true) -> true
(false || false) -> false
```

5. The following is a Java example of using the || (or) operator.

```
((2+3 < 10) || (19 > 21)) -> true
```

The || operator also performs short-circuit evaluation in Java. If the first half of an || statement is true, the operator immediately returns true without evaluating the second half.

6. The ! operator is a unary operator that changes a **boolean** value to its opposite.

```
(! false == true) -> true
(! true == false) -> true
(! true == true) -> false
!(2 < 3) -> false
```

F. Precedence and Associativity of Operators

page 8 of 17

1. Introducing two new sets of operators (relational and logical) adds to the complexity of operator precedence in Java. An abbreviated precedence chart is included here.

Operator	Associativity
! unary - ++ --	right to left
* / %	left to right
+ -	left to right
< <= > >=	left to right
== !=	left to right

&& (and)	left to right
(or)	left to right
= += -= *= /=	right to left

Table 8-1 Precedence and Associativity of Operators

- Because the logical operators have low precedence in Java, parentheses are not needed to maintain the correct order of solving problems. However, they can be used to make complex expressions more readable.

```
((2 + 3 < 10) && (75 % 12 != 12)) // easier to read
(2 + 3 < 10 && 75 % 12 != 12)      // harder to read
```

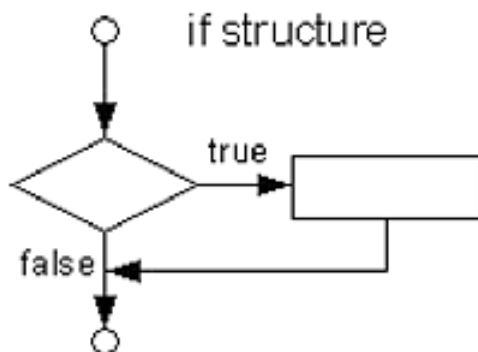
G. The if-else Statements

page 9 of 17

- The general syntax of the if statement is as follows:

```
if (expression){
statement1;
}
```

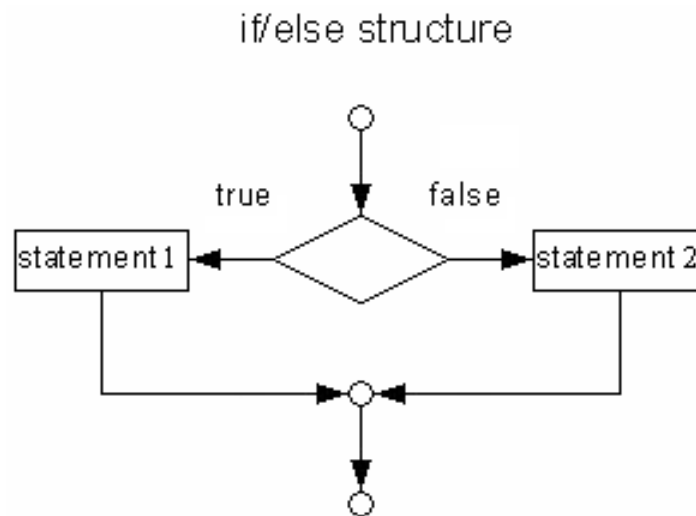
If the expression evaluates to true, statement1 is executed. If expression is false then nothing is executed and the program execution picks up after the ending curly brace `}`. The following diagram shows the flow of control:



- To provide for two-way selection an if statement may add an else option.

```
if (expression){
statement1;
}else{
statement2;
}
```

If the expression evaluates to true, the statement is executed. In an if-else statement, if the expression is false then statement2 would be executed. The following flowchart illustrates the flow of control.



3. The expression being tested must always be placed in parentheses. This is a common source of syntax errors.

H. Compound Statements

page 10 of 17

1. The statement executed in a control structure can be a block of statements, grouped together into a single compound statement.
2. A compound statement is created by enclosing any number of single statements by braces as shown in the following example:

```
if (expression){  
statement1;  
statement2;  
statement3;  
}else{  
statement4;  
statement5;  
statement6;  
}
```

I. Nested if-else Statements

page 11 of 17

1. The statement inside of an `if` or `else` option can be another `if-else` statement. Placing an `if-else` inside another is known as nested `if-else` constructions. For example:

```
if (expression1){
    if (expression2){
        statement1;
    }else{
        statement2;
    }
}else{
    statement3;
}
```

2. Here, your braces will need to be correct to ensure that the ifs and elses get paired with their partners.
3. The above example has three possible different outcomes as shown in the following chart:
4. Technically, braces are not needed for `if` and `if-else` structures if you only want one statement to execute. However, caution must be shown when using `else` statements inside of nested `if-else` structures. For example:

```
if (expression1)
    if (expression2)
        statement1;
else
    statement2;
```

Indentation is ignored by the compiler, hence it will pair the `else` statement with the inner `if`. If you want the `else` to get paired with the outer `if` as the indentation indicates, you need to add braces:

```
if (expression1){
    if (expression2)
        statement1;
}else
    statement2;
```

The braces allow the `else` statement to be paired with the outer `if`.

Important Concept	However, if you always use braces when writing <code>if</code> and <code>if-else</code> statements, you will never have this problem.
--------------------------	---

5. Another alternative to the example in Section 4 makes use of the `&&` operator. A pair of nested if statements can be coded as a single compound `&&` statement. Both of these blocks of code would have the exact same effect, but the second one is slightly easier to read.

```
if(expression1){
    if(expression2){
        statement1;
    }
}

//or...

if (expression1 && expression2){
    statement1;
}
```

The second block of code makes the conditions clearer to another programmer.

6. Consider the following example of determining the type of triangle given the three sides A, B, and C.

```
if ( (A == B) && (B == C) )
    System.out.println("Equilateral triangle");
else if ( (A == B) || (B == C) || (A == C) )
    System.out.println("Isosceles triangle");
else
    System.out.println("Scalene triangle");
```

If an equilateral triangle is encountered, the rest of the code is ignored. This can help to reduce the execution time of a program.

J. Conditional Operator

page 12 of 17

1. Java provides an alternate method of coding an `if-else` statement using the conditional operator. This operator is the only ternary operator in Java, as it requires three operands. The general syntax is:

```
(condition) ? statement1 : statement2;
```

2. If the condition is true, `statement1` is executed. If the condition is false, `statement2` is executed.

3. This is appropriate in situations where the conditions and statements are fairly compact.

```
int max(int a, int b){ // returns the larger of two integers
    (a > b) ? return a : return b;
}
```

K. Boolean Identifiers

page 13 of 17

1. The execution of **if-else** statements depends on the value of the Boolean expression. We can use **boolean** variables to write code that is easier to read.
2. For example, the **boolean** variable **done** could be used to write code that reads more like English.

Instead of

```
if(done == true){
    System.out.println("We are done!");
}
```

we can write

```
if(done){
    System.out.println("We are done!");
}
```

3. Programmers often use **boolean** variables to aid in program flow and readability. The second version is the more preferred way of using a **boolean** variable in this situation because it is less dangerous. If you make a mistake and only put **=** instead of **==** Java will not catch that and interprets the statement as assignment. Some strange results could occur and it can take the programmer a while to catch the error.

L. Switch Statements (Optional)

page 14 of 17

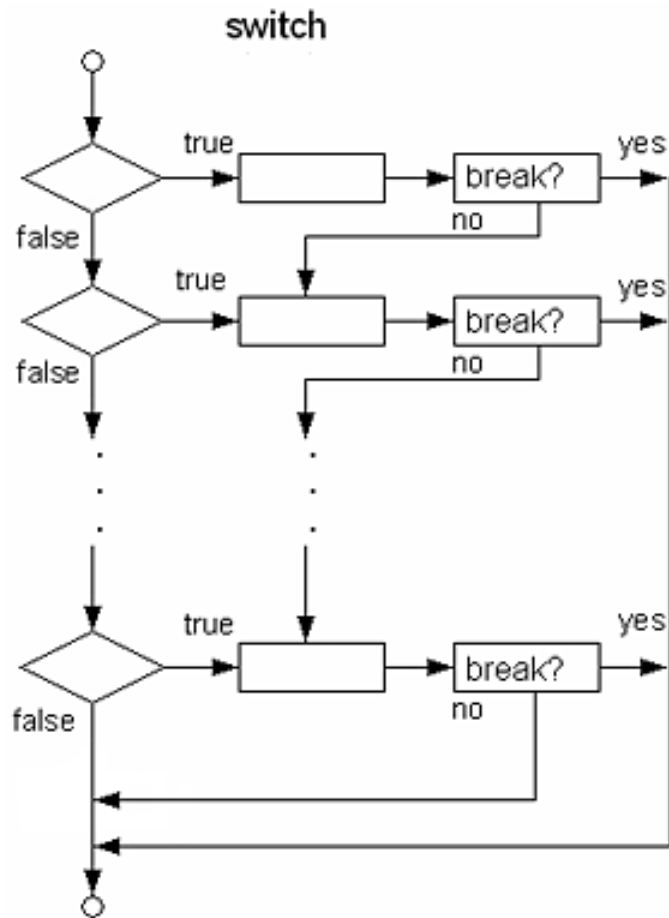
1. Consider a simple user menu for a store simulation program. There should be options

to buy certain items, check your total money spent, cancel items selected, exit, and finish and pay. We could take the input from this menu and do a complicated, nested series of **if-else** statements, but that would quickly become bulky and difficult to read. However, there is an easy way to handle such data input with a **switch** statement. Depending on which command is chosen, the program will select one direction out of many. The AP exam does not test on the **switch** statement, but we include it here at the end of this chapter because it is a very useful tool to have in your programming toolkit.

2. The general form of a **switch** statement is:

```
switch (expression){  
    case value1:  
        statement1;  
        statement2;  
        ...  
        break;  
    case value2:  
        statement3;  
        statement4;  
        ...  
        break;  
    case valuen: //any number of case statement may be  
used  
        statement;  
        statement;  
        break;  
    default:  
        statement;  
        statement;  
        break;  
}          /* end of switch statement */
```

3. The flow of control of a **switch** statement is illustrated in this diagram:



4. The **switch** statement attempts to match the integer value of the expression with one of the **case** values.
5. If a match occurs, then all statements past the **case** value are executed until a **break** statement is encountered.
6. The effect of the break statement causes program control to jump to the end of the switch statement. No other cases are executed.
7. A very common error when coding a **switch** control structure is forgetting to include the **break** statements to terminate each case value. If the **break** statement is omitted, all the statements following the matching **case** value are executed. This is usually very undesirable.
8. If it is possible that none of the case statements will be true, you can add a **default** statement at the end of the **switch**. This will only execute if none of the case statements happened. If all possibilities are covered in your case statements, the **default** statement is unnecessary. Note that the **default** statement can actually be placed anywhere. If you place the **default** in the beginning or middle of the **switch**, you will probably want to end the **default** case with a **break**. Otherwise, execution will continue with the case after the **default**.

```

int i = 4;
switch (i) {
    case 1: System.out.println("Apple"); break;
    default: System.out.println("Orange");
    case 2: System.out.println("Banana"); break;
}

```

Orange
Banana

9. The following example applies the switch statement to printing the work day of the week corresponding to a value. We pass in the integer day:

```

switch (day){
    case 1: System.out.println ("Monday"); break;
    case 2: System.out.println ("Tuesday"); break;
    case 3: System.out.println ("Wednesday"); break;
    case 4: System.out.println ("Thursday"); break;
    case 5: System.out.println ("Friday"); break;
    default: System.out.println ("not a valid day"); break;
}

```

10. Suppose we wanted to count the occurrences of vowels and consonants in a stream of text.

```

if (('a' <= letter) && (letter <= 'z')){
    switch (letter){
        case 'a' : case 'e' : case 'i' : case 'o' : case 'u' :
            vowel++;
            break;
        default :
            consonant++;
            break;
    }
}

```

- a. Note that multiple case values can lead to one set of statements.
 - b. It is good programming practice to include a break statement at the end of the switch structure. If you need to go back and add another case statement at the end of the switch structure, a break statement already terminates the previous case statement and there is no chance that you might forget to add a break statement.
11. There are programming situations where the switch statement should not replace an if-else chain. If the value being compared must fit in a range of values, the if-else statement should be used.


```
if(score >= 90 && score <= 100){  
    grade = 'A';  
} else if( (score >= 80 && score < 90)  
    grade = 'B';  
} else if( (score >= 70 && score < 80)  
    grade = 'C';  
}
```

etc...

You should not replace the above structure with a switch statement.

12. Finally, the **switch** statement cannot compare **double** values.

Summary/Review

page 15 of 17

Control structures are a fundamental part of Java. You will need to practice control structures in Java to become familiar with what types of situations they are useful in. Also, Boolean expressions are very useful and should be used whenever appropriate to make coding easier.

Chapter 9

A9 Introduction

page 1 of 8

Recursion is the process of a method calling itself as part of the solution to a problem. It is a problem solving technique that can turn long and difficult solutions into compact and elegant answers.

The key topics for this lesson are:

- A. [Recursion](#)
- B. [Pitfalls of Recursion](#)

C. [Recursion Practice](#)

[A9 Vocabulary](#)

page 2 of 8

BASE CASE
STACK

RECURSION
STACK OVERFLOW ERROR

A. Recursion

page 3 of 8

1. Recursion occurs when a method calls itself to solve another version of the same problem. With each recursive call, the problem becomes simpler and moves towards a base case. A base case is when the solution to the problem can be calculated without another recursive call.
2. Recursion involves the internal use of a stack. A stack is a data abstraction that works like this: New data is "pushed," or added to the top of the stack. When information is removed from the stack it is "popped," or removed from the top of the stack. The recursive calls of a method will be stored on a stack and manipulated in a similar manner.
3. The problem of computing factorials is our first example of recursion. The factorial operation in mathematics is illustrated below.

$$1! = 1$$

$$2! = 2 * 1 \quad \text{or} \quad 2 * 1!$$

$$3! = 3 * 2 * 1 \quad \text{or} \quad 3 * 2!$$

$$4! = 4 * 3 * 2 * 1 \quad \text{or} \quad 4 * 3!$$

Notice that each successive line can be solved in terms of the previous line. For example, $4!$ is equivalent to

$$4 * 3!$$

A recursive method to solve the factorial problem is given below. Notice the recursive call in the last line of the method. The method calls another implementation of itself to solve a smaller version of the problem.

```
int fact(int n){
// returns the value of n!
// precondition: n >= 1
    if (n == 1){
        return 1;
    }else{
        return n * fact(n - 1);
    }
}
```

4. The base case is a fundamental situation where no further problem solving is necessary. In the case of finding factorials, $1!$ is by definition 1. No further work is needed. Each recursive method must have at least one base case.
5. Suppose we call the method to solve `fact(4)`. This will result in four calls of method `fact`.
6. When a recursive call is made, the current computation is temporarily suspended and placed on the stack with all its current information available for later use.
7. A completely new copy of the method is used to evaluate the recursive call. When that is completed, the value returned by the recursive call is used to complete the suspended computation. The suspended computation is removed from the stack and its work now proceeds.
8. When the base case is encountered, the recursion will now unwind and result in a final answer. The expressions below should be read from right to left.

$$\text{fact}(4) = 4 * \text{fact}(3) = 3 * \text{fact}(2) = 2 * \text{fact}(1) = 1$$

$$24 \leftarrow 4 * 6 \leftarrow 3 * 2 \leftarrow 2 * 1$$

Figure 9.1 below diagrams what happens:

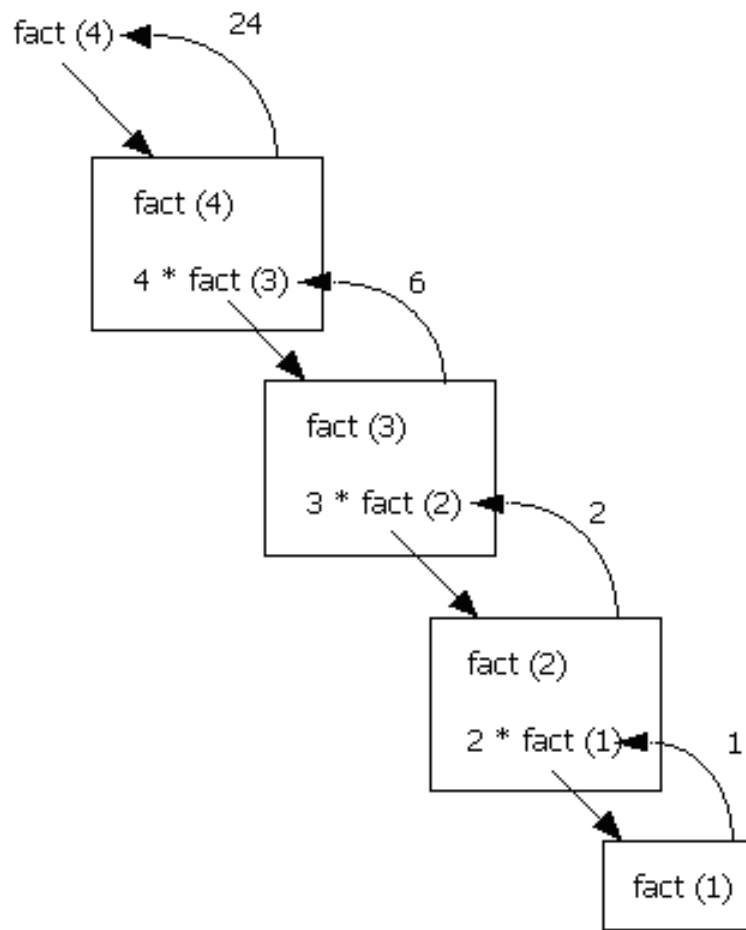


Figure 9.1 - Recursive Boxes

Each box represents a call of method `fact`. To solve `fact(4)` requires four calls of method `fact`.

9. Notice that when the recursive calls were made inside the `else` statement, the value fed to the recursive call was $(n-1)$. This is where the problem is getting simpler with the eventual goal of solving `1!`.

B. Pitfalls of Recursion

page 4 of 8

1. If the recursion never reaches a base case, the recursive calls will continue until the computer runs out of memory and the program crashes. Experienced programmers try to examine the remains of a crash. The message “stack overflow error” or “heap storage exhaustion” indicates a possible runaway recursion.
2. When programming recursively, you need to make sure that the algorithm is moving toward a base case. Each successive call of the algorithm must be solving a version

of the problem that is closer to a base case.

C. Recursion Practice

page 5 of 8

1. To provide some practice, write a recursive power method that raises a base to some exponent, n . Use integers to keep things simple.

```
double power(int base, int n){
// Recursively determines base raised to
// the nth power. Assumes 0 <= n <= 10.

}

```

Summary/Review

page 6 of 8

Recursion takes some time and practice to get used to. Eventually, you want to be able to think recursively without the aid of props and handouts. Study the examples provided in these notes and work it through for yourself. Recursion is a very powerful programming tool for solving difficult problems.

Chapter 10

A10 Introduction

page 1 of 17

Strings are needed in many programming tasks. Much of the information that identifies a person must be stored as a string: name, address, city, social security number, etc. This lesson covers the specifications of the String class and how to use it to solve string-processing problems.

The key topics for this lesson are:

- A. [The String Class](#)
- B. [String Constructors](#)
- C. [Object References](#)
- D. [The null Value](#)
- E. [String Query Methods](#)
- F. [String Translation Methods](#)
- G. [Immutability of Strings](#)
- H. [Comparing Strings](#)
- I. [Strings and Characters](#)
- J. [The toString Method](#)
- K. [String I/O](#)

[A10 Vocabulary](#)

page 2 of 17

charAt	compareTo
CONCATENATION	equals
GARBAGE	GARBAGE COLLECTION
IMMUTABLE	length
nextLine	next
STRING CLASS	null
substring	STRING LITERAL
toString	toLowerCase
trim	toUpperCase

A. The String Class

page 3 of 17

1. Groups of characters in Java are not represented by primitive types as are `int` or `char` types. Strings are objects of the `String` class. The `String` class is defined in `java.lang.String`, which is automatically imported for use in every program you write.

We've used String literals, such as "Enter a value" with `System.out.print` statements in earlier examples. Now we can begin to explore the `String` class and the capabilities that it offers.

2. So far, our experience with Strings has been with String literals, consisting of any sequence of characters enclosed within double quotation marks. For example:

```
"This is a string"  
"Hello World!"  
"\tHello World!\n"
```

The characters that a `String` object contains can include escape sequences. This example contains a tab (`\t`) and a linefeed (`\n`) character.

3. A second unique characteristic of the `String` class is that it supports the "+" operator to concatenate two `String` expressions. For example:

```
sentence = "I " + "want " + "to be a " + "Java programmer.";
```

The "+" operator can be used to combine a `String` expression with any other expression of primitive type. When this occurs, the primitive expression is converted to a `String` representation and concatenated with the string. For example, consider the following instruction sequence:

```
PI = 3.14159;  
System.out.println("The value of PI is " + PI);
```

Run Output:

```
The value of PI is 3.14159
```

To invoke the concatenation, at least one of the items must be a `String`.

B. String Constructors

page 4 of 17

1. Because Strings are objects, you can create a `String` object by using the keyword `new` and a `String` constructor method, just as you would create any other object.

```
String name = new String();  
String name2 = new String("Nancy");
```

2. Though they are not primitive types, strings are so important and frequently used that

Java provides additional syntax for declaration:

```
String aGreeting = "Hello world";
```

A `String` created in this short-cut way is called a *String literal*. Only `Strings` have a shortcut like this. All other objects are constructed by using the `new` operator.

Many new Java programmers get confused because of this shortcut and believe that `Strings` are primitive data types. However, `Strings` are objects and therefore have behaviors and attributes.

C. Object References

page 5 of 17

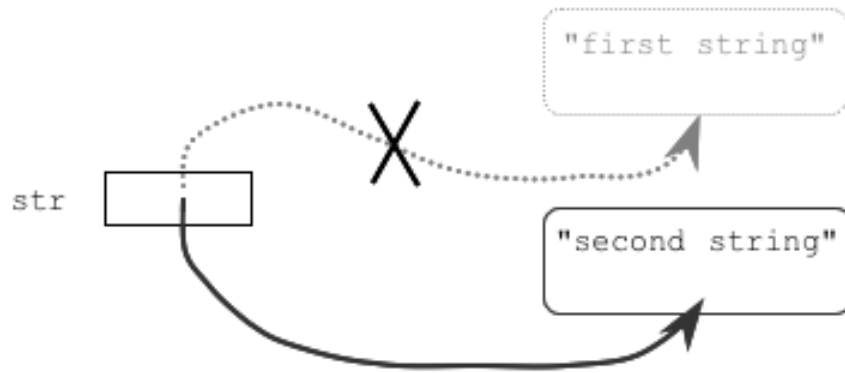
1. Recall from Lesson A2 that an object is constructed as an instance of a particular class. The object is most often referenced using an identifier. The identifier is a variable that stores the reference to the object. This identifier is called an object reference. Now that we are working with a simple class, `String`, it is a good time to discuss object references.
2. Whenever the `new` operator is used, a new object is created. Each time an object is created, there is a reference to where it is stored in memory. The reference can be saved in a variable. The reference is used to find the object.
3. It is possible to store a new object reference in a variable. For example:

```
String str;  
  
str = new String("first string");  
System.out.println(str);  
  
str = new String("second string");  
System.out.println(str);
```

Run Output:

```
first string  
second string
```

In the example above, the variable `str` is used to store a reference to the `String`, “first string”. In the second part of the example a reference to the `String`, “second string” is stored in the variable `str`. If another reference is saved in the variable, it *replaces* the previous reference (see diagram below).



4. If a reference to an object is no longer being used then there is no way to find it, and it becomes "*garbage*." The word "garbage" is the correct term from computer science to use for objects that have no references. This is a common situation when new objects are created and old ones become unneeded during the execution of a program. While a program is running, a part of the Java system called the "garbage collector" reclaims each lost object (the "garbage") so that the memory is available again. In the above example, the `String` object "first string" becomes garbage.
5. Multiple objects of the same class can be maintained by creating unique reference variables for each object.

```
String strA; // reference to the first object
String strB; // reference to the second object

// create the first object and save its reference
strA = new String("first string");

// print data referenced by the first object.
System.out.println(strA);

// create the second object and save its reference
strB = new String("second string");

// print data referenced by the second object.
System.out.println(strB);

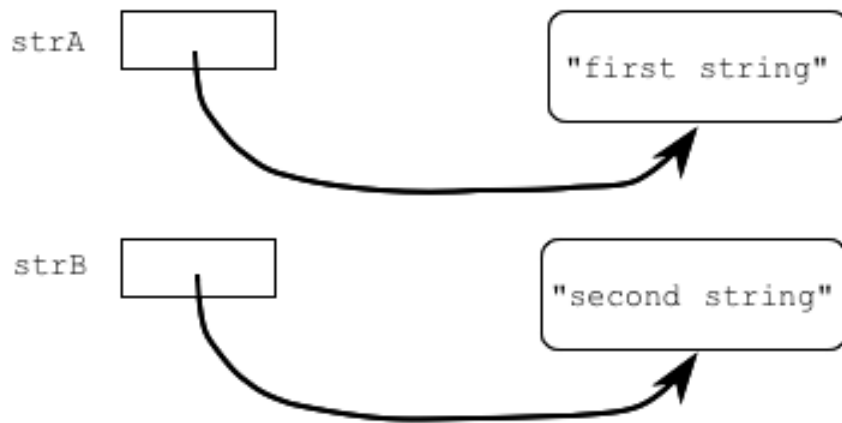
// print data referenced by the first object.
System.out.println(strA);
```

Run Output:

```
first string
second string
first string
```

This program has two reference variables, `strA` and `strB`. It creates two objects and places each reference in one of the variables. Since each object has its own reference

variable, no reference is lost, and no object becomes garbage (until the program has finished running).



6. Different reference variables that refer to the same object are called aliases. In effect, there are two names for the same object. For example:

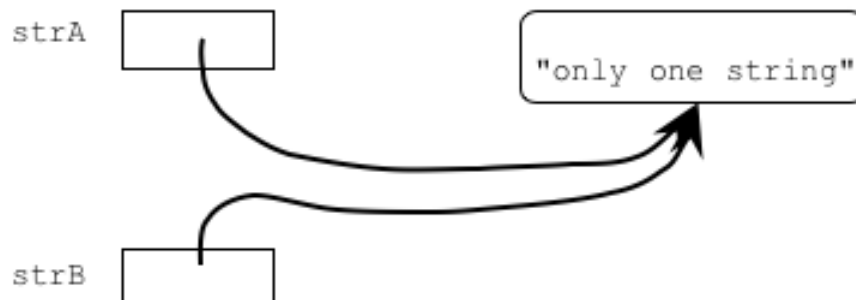
```
String strA; // reference to the object
String strB; // another reference to the object

// Create the only object and save its
// reference in strA
strA = new String("only one string");
System.out.println(strA);

strB = strA; // copy the reference to strB.
System.out.println(strB);
```

Run Output:

```
only one string
only one string
```



When this program runs, only one object is created (by `new`). Information about how to find the object is put into `strA`. The assignment operator in the statement

```
strB = strA; // copy the reference to strB
```

copies the information that is in `strA` to `strB`. It does not make a copy of the object.

D. The null Value

page 6 of 17

1. In most programs, objects are created and objects are destroyed, depending on the data and on what is being computed. A reference variable sometimes does and sometimes does not refer to an object. You may need a way to erase the reference inside a variable without creating a new reference. You do this by assigning `null` to the variable.
2. The value `null` is a special value that means "no object." A reference variable is set to `null` when it is not referring to any object.

```
String a =          // 1. an object is created;
    new String("stringy"); // variable a refers to it
String b = null;    // 2. variable b refers to no
                   // object.
String c =          // 3. an object is created
    new String(""); // (containing no characters)
                   // variable c refers to it
if (a != null)      // 4. statement true, so
    System.out.println(a); // the println(a) executes.

if (b != null)      // 5. statement false, so the
    System.out.println(b); // println(b) is skipped.

if (c != null)      // 6. statement true, so the
    System.out.println(c); // println(c) executes (but
                           // it has no characters to
                           // print).
```

Run Output:

stringy

3. Variables `a` and `c` are initialized to object references. Variable `b` is initialized to `null`. Note that variable `c` is initialized to a *reference* to a `String` object containing no characters. Therefore `println(c)` executes, but it has no characters to print. Having no characters is different from the value being `null`.

Query Method	Sample Syntax
<code>int length();</code>	<pre>String str1 = "Hello!"; int len = str1.length(); // len == 6</pre>
<code>char charAt(int index);</code>	<pre>String str1 = "Hello!"; char ch = str1.charAt(0); // ch == 'H'</pre>
<code>int indexOf(String str);</code>	<pre>String str2 = "Hi World!"; int n = str2.indexOf("World"); // n == 3 int n = str2.indexOf("Sun"); // n == -1</pre>
<code>int indexOf(char ch);</code>	<pre>String str2 = "Hi World!"; int n = str2.indexOf('!'); // n == 8 int n = str2.indexOf('T'); // n == -1</pre>

1. The `int length()` method returns the number of characters in the `String` object.
2. The `charAt` method is a tool for extracting a character from within a `String`. The `charAt` parameter specifies the position of the desired character (0 for the leftmost character, 1 for the second from the left, etc.). For example, executing the following two instructions prints the `char` value 'X'.

```
String stringVar = "VWXYZ";
System.out.println(stringVar.charAt(2));
```

3. The `int indexOf(String str)` method will find the first occurrence of `str` within this `String` and return the index of the first character. If `str` does not occur in this `String`, the method returns -1.
4. The `int indexOf(char ch)` method is identical in function and output to the other `indexOf` function except it is looking for a single character.

Translate Method	Sample Syntax
<code>String toLowerCase();</code>	<code>String greeting = "Hi World!";</code>

	<pre>greeting = greeting.toLowerCase(); // greeting <- "hi world!"</pre>
String toUpperCase();	<pre>String greeting = "Hi World!"; greeting = greeting.toUpperCase(); // greeting <- "HI WORLD!"</pre>
String trim();	<pre>String needsTrim = " trim me!"; needsTrim = needsTrim.trim(); // needsTrim <- "trim me!"</pre>
String substring(int beginIndex)	<pre>String sample = "hamburger"; sample = sample.substring(3); // sample <- "burger"</pre>
String substring(int beginIndex, int endIndex)	<pre>String sample = "hamburger"; sample = sample.substring(4, 8); // sample <- "urge"</pre>

1. toLowerCase() returns a String with the same characters as the String object, but with all characters converted to lowercase. Notice that in all of the above samples, the String object is placed on the left hand side of the assignment statement. This is necessary because Strings in Java are immutable. Please see section G for a full explanation of immutable.
2. toUpperCase() returns a String with the same characters as the String object, but with all characters converted to uppercase.
3. trim() returns a String with the same characters as the String object, but with the leading and trailing whitespace removed.
4. substring(int beginIndex) returns the substring of the String object starting from beginIndex through to the end of the String object.
5. substring(int beginIndex, int endIndex) returns the substring of the String object starting from beginIndex through, but not including, position endIndex of the String object. That is, the new String contains characters numbered beginIndex to endIndex-1 in the original String.

Immutability of strings means you cannot modify any string object.

Notice the above example for the method `toLowerCase`. This method returns a new string, which is the lower case version of the object that invoked the method.

```
String greeting = "Hi World!";  
greeting.toLowerCase();  
System.out.println(greeting);
```

Run Output:

Hi World!

The object `greeting` did not change. To change the value of `greeting`, you need to assign the return value of the method to the object `greeting`.

```
greeting = greeting.toLowerCase();  
System.out.println(greeting);
```

Run Output:

hi world!

H. Comparing Strings

page 10 of 17

1. The following methods should be used when comparing String objects:

Comparison Method	Sample Syntax
boolean equals(String anotherString);	String aName = "Mat"; String anotherName = "Mat"; if (aName.equals(anotherName)) System.out.println("the same");
boolean equalsIgnoreCase(String anotherString);	String aName = "Mat"; if (aName.equalsIgnoreCase("MAT")) System.out.println("the same");
int compareTo(String anotherString)	String aName = "Mat" n = aName.compareTo("Rob"); // n < 0 n = aName.compareTo("Mat"); // n == 0 n = aName.compareTo("Amy"); // n >

2. The `equals()` method evaluates the contents of two `String` objects to determine if they are equivalent. The method returns `true` if the objects have identical contents. For example, the code below shows two `String` objects and several comparisons. Each of the comparisons evaluate to `true`; each comparison results in printing the line "Name's the same".

```
String aName = "Mat";
String anotherName = new String("Mat");

if (aName.equals(anotherName))
    System.out.println("Name's the same");

if (anotherName.equals(aName))
    System.out.println("Name's the same");

if (aName.equals("Mat"))
    System.out.println("Name's the same");
```

Each `String` shown above, `aName` and `anotherName`, is an object of type `String`, so each `String` has access to the `equals()` method. The `aName` object can call `equals()` with `aName.equals(anotherName)`, or the `anotherName` object can call `equals()` with `anotherName.equals(aName)`. The `equals()` method can take either a variable `String` object or a literal `String` as its argument.

In all three examples above, the boolean expression evaluates to `true`.

3. The `==` operator can create some confusion when comparing objects. The `==` operator will check the reference value, or address, of where the object is being stored. It will not compare the data members of the objects. Because `Strings` are objects and not primitive data types, `Strings` cannot be compared with the `==` operator. However, due to the shortcuts that make `String` act in a similar way to primitive types, two `Strings` created without the `new` operator but with the same `String` literal will actually point to the same address in memory. Observe the following code segment and its output:

```
String aGreeting1 = new String("Hello");
String anotherGreeting1 = new String("Hello");

if (aGreeting1 == anotherGreeting1)
    System.out.println("This better not work!");
else
    System.out.println("This prints since each object " +
        "reference is different.");

String aGreeting2 = "Hello";
String anotherGreeting2 = "Hello";
```

```

if (aGreeting2 == anotherGreeting2)
    System.out.println("This prints since both " +
        "object references are the same!");
else
    System.out.println("This does not print.");

```

Run Output:

This prints since each object reference is different.
 This prints since both object references are the same!

The objects `aGreeting1` and `anotherGreeting1` are each instantiated using the `new` command, which assigns a different reference to each object. The `==` operator compares the reference to each object, not their contents. Therefore the comparison `(aGreeting1 == anotherGreeting1)` returns **false** since the references are different.

The objects `aGreeting2` and `anotherGreeting2` are String literals (created without the `new` command - i.e. using the short-cut instantiation process unique to Strings). In this case, Java recognizes that the contents of the objects are the same and it creates only one instance, with `aGreeting2` and `anotherGreeting2` each referencing that instance. Since their references are the same, `(aGreeting2 == anotherGreeting2)` returns **true**.

4. When comparing objects to see if they are equal, always use the `equals` method. It would be a rare occasion to care if they are occupying the same memory location. Remember that a `String` is an object!
5. The `equalsIgnoreCase()` method is very similar to the `equals()` method. As its name implies, it ignores case when determining if two `Strings` are equivalent. This method is very useful when users type responses to prompts in your program. The `equalsIgnoreCase()` method allows you to test entered data without regard to capitalization.
6. The `compareTo()` method compares the calling `String` object and the `String` argument to see which comes first in the lexicographic ordering. Lexicographic ordering is the same as alphabetical ordering when both strings are either all uppercase or all lowercase. If the calling string is first lexicographically, it returns a negative value. If the two strings are equal, it returns zero. If the argument string comes first lexicographically, it returns a positive number.

```

String bob = "Bob";
String bob2 = "bob";
String steve = "Steve";
System.out.println(bob.compareTo(bob2));
System.out.println(bob2.compareTo(bob));

```



```
System.out.println(steve.compareTo(bob2));  
System.out.println(bob.compareTo(steve));
```

The output for this block of code would be:

```
-32  
32  
-15  
-17
```

I. Strings and Characters

page 11 of 17

1. It is natural to think of a **char** as a **String** of length 1. Unfortunately, in Java the **char** and **String** types are incompatible since a **String** is an object and a **char** is a primitive type. This means that you cannot use a **String** in place of a **char** or use a **char** in place of a **String**.
2. Extracting a **char** from within a **String** can be accomplished using the `charAt` method as previously described.
3. Conversion from **char** to **String** can be accomplished by using the "+" (concatenation) operator described previously. Concatenating any **char** with an *empty string* (**String** of length zero) results in a **String** that consists of that **char**. The java notation for an empty string is two consecutive double quotation marks. For example, to convert `myChar` to a **String** it is added to "".

```
char myChar = 'X';  
String myString = "" + myChar;  
System.out.println(myString);  
char anotherChar = 'Y';  
myString += anotherChar;  
System.out.println(myString);
```

The output of this block of code would be:

```
X  
XY
```

1. Wouldn't it be nice to be able to output objects that you have made using the simple line `System.out.print(Object name)`? Let's consider the example of the `RegularPolygon` class discussed in Lesson A6. It would be nice to be able to print out the statistics of your `RegularPolygon` objects without having to do a lot of `System.out.print` statements. Thanks to the `toString` method, you have the ability to do this.
2. You can create a `toString` method in any of your classes in the format of `public String toString()`. Within the `toString()` method, you can format your class variables into one `String` object and return that `String`. Then, when Java encounters your `Object` in a `String` format, it will call the `toString()` method. Let's look at an example using a `RegularPolygon` class.

```
public String toString(){
String a = "Sides: " + getSides();
a += " Length: " + getLength();
a += " Area: " + getArea();
return a;
}
```

```
RegularPolygon square = new RegularPolygon(4, 10);
System.out.println(square);
```

Run Output:

Sides: 4 Length: 10 Area: 100

3. You must be careful when using this, because you are fixing the format of the output. Oftentimes, you will still want to format your output depending on the specific problem you are solving, but the `toString()` method provides a simple and quick way to look at the state of your objects. There are also many times when the `toString()` method will be very useful. Consider a `Student` class that contains member variables for first name, middle name, last name, a list of classes being taken, the student's address and phone number, etc. You could easily make a `toString()` method that would simply output the students first name, middle initial, and last name for quick reference. Every time you design a class, you should stop and think about whether or not your class would benefit from having a `toString()` method and how you should format this `String`.

1. The scanner class has two methods for reading textual input from the keyboard.
2. The `next` method returns a reference to a `String` object that has from zero to many characters typed by the user at the keyboard. The `String` will end whenever it reaches white space. White space is defined as blank spaces, tabs, or newline characters in the input stream. When inputting from the keyboard, `next` stops adding text to the `String` object when the first white space is encountered from the input stream.
3. A `nextLine` method returns a reference to a `String` object that contains from zero to many characters entered by the user. With `nextLine`, the `String` object may contain blank spaces and tabs but will end when it reaches a newline character. Therefore, `nextLine` will read in whole lines of input rather than only one word at a time.
4. String input is illustrated below.

```
Scanner keyboard = new Scanner(System.in);
String word1, word2, anotherLine;

// prompt for input from the keyboard
System.out.print("Enter a line: ");

// grab the first "word"
word1 = keyboard.next();

// grab the second "word"
word2 = keyboard.next();

// prompt for input from the keyboard
System.out.print("Enter another line: ");

// discard any remaining input from previous line
// and read the next line of input
anotherLine = keyboard.nextLine(); //skip to the next line
anotherLine = keyboard.nextLine(); //grab all of the next line

// output the strings
System.out.println("word1 = " + word1);
System.out.println("word2 = " + word2);
System.out.println("anotherLine = " + anotherLine);
```

Run Output:

```
Enter a line: Hello World! This will be discarded.
Enter another line: This line includes whitespace.
word1 = Hello
word2 = World!
anotherLine = This line includes whitespace.
```

5. Formatting Strings is done with the same style as using the `printf()` method discussed in Lesson A7, Simple I/O. In fact, now that you know more about Strings, you should be able to recognize that you are really manipulating String literals when you use the `printf()` formatting rules. If you want to alter how a String object is stored without actually printing it to the String, you can simply use a `Formatter` object (which is actually the object that `printf()` uses itself). An example of how to use `Formatter` is shown below. Note: Don't forget to import the `Formatter` class.

```
import java.util.Formatter;

Formatter f = new Formatter();
f.format("%10s", "Bob");
String bob = f.toString();
System.out.println(bob.length());
System.out.println(bob);
```

Run Output:

```
10
    Bob
```

LAB ASSIGNMENT A10.3

page 17 of 17

RomanNumerals

Assignment:

1. You will create a class with two `public static` methods. One will receive an `int` number and return a `String` with that `int` converted into Roman Numerals. The other method will receive a `String` of Roman Numerals and return the `int` value of the Roman Numerals.
2. These two methods are `static` because there is no reason to create an object just to run these calculations.
3. Think of using helper methods to reuse algorithms needed to solve these problems.
4. Assume that your client gives you valid Roman Numerals and the Arabic numbers are positive and less than 4000.

Instructions:

Roman Numerals work differently than our normal Arabic number system. Roman Numerals have symbols, all in capital letters (and sometimes in lower case), which represent Arabic numbers. Roman Numerals have been used for identifying movie sequels (i.e., *The Godfather: Part II*), for publication copyright dates, for numbering monarchs such as Queen Elizabeth II, and for numbering Super Bowls. See the following table for the Roman Numerals symbols up to 1000.

Roman Numeral	Arabic Number
I or i	1
V	5
X	10
L	50
C	100
D	500
M	1000

Usually, numbers are formed by stringing the Roman numerals together and adding them up to make the required number (i.e., II = 2, or XII = 12). If smaller numbers follow larger numbers, the numbers are added (i.e., VIII = 5 + 3 or 8), but if a smaller number precedes a larger number, the smaller number is subtracted from the larger number (i.e., IX = 10 - 1 or 9).

There is shorthand for the case when there are four of the same symbols in a row. Instead of IIII for 4, it is written as IV or 5 - 1 = 4. This only applies to symbols that represent powers of ten. Since our numbers will be less than 4000, this only makes sense for I, X and C. Some people think this means you can write IC for 99 but that is not going to be allowed. When using this shortcut, a symbol can only precede a symbol whose value is 5 or 10 times its own value. For example, X (10) can only precede L (50) or C (100). So XL (40) is acceptable, but XD (490?) is not.

Roman Numeral	Arabic Number
XLVI	46
XCIX	99
MDCCCXIX	1819
DCXLIX	649
MCMLXXXIII	1983

Recursion takes some time and practice to get used to. Eventually, you want to be able to think recursively without the aid of props and handouts. Study the examples provided in these notes and work it through for yourself. Recursion is a very powerful programming tool for solving difficult problems.

Chapter 11

A11 Introduction

page 1 of 10

Inheritance, a major component of OOP, is a technique that will allow you to define a very general class and then later define more specialized classes based upon it. You will do this by adding some new capabilities to the existing class definitions or changing the way the existing methods work. Inheritance saves work because the more specialized class inherits all the properties of the general class and you, the programmer, only need to program the new features.

The key topics for this lesson are:

- A. [Single Inheritance](#)
- B. [Class Hierarchies](#)
- C. [Using Inheritance](#)
- D. [Method Overriding](#)
- E. [Interfaces](#)

[A11 Vocabulary](#)

page 2 of 10

BASE CLASS

CHILD CLASS

Created by mu

DERIVED CLASS

implements

METHOD OVERRIDING

SUBCLASS

SUPERCLASS

extends

interface

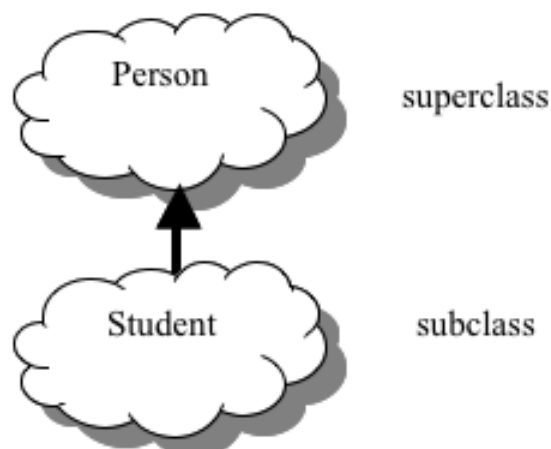
PARENT CLASS

super

A. Single Inheritance

page 3 of 10

1. *Inheritance* enables you to define a new class based on a class that already exists. The new class will inherit the characteristics of the existing class, but may also provide some additional capabilities. This makes programming easier, because you can reuse and extend your previous work and avoid duplication of code.
2. The class that is used as a basis for defining a new class is called a *superclass* (or *parent class* or *base class*). The new class based on the superclass is called a *subclass* (or *child class* or *derived class*.)
3. The process by which a subclass inherits characteristics from just one parent class is called single inheritance. Some languages allow a derived class to inherit from more than one parent class in a process called multiple inheritance. Multiple inheritance makes it difficult to determine which class will contribute what characteristics to the child class. Java avoids these issues by only providing support for single inheritance.
4. Figure 11.1 shows a superclass and a subclass. The line between them shows the "is a" relationship. The picture can be read as "a Student is a Person." The clouds represent the classes. That is, the picture does not show any particular Student or any particular Person, but shows that the class Student is a subclass of the Person class.



Created by mu

Figure 11.1 - Subclass and Superclass

5. Inheritance is between classes, not between objects. A superclass is a blueprint that is followed when a new object is constructed. That newly constructed object is another blueprint that looks much like the original, but with added features. The subclass in turn can be used to construct objects that look like the superclass's objects, but with additional capabilities.

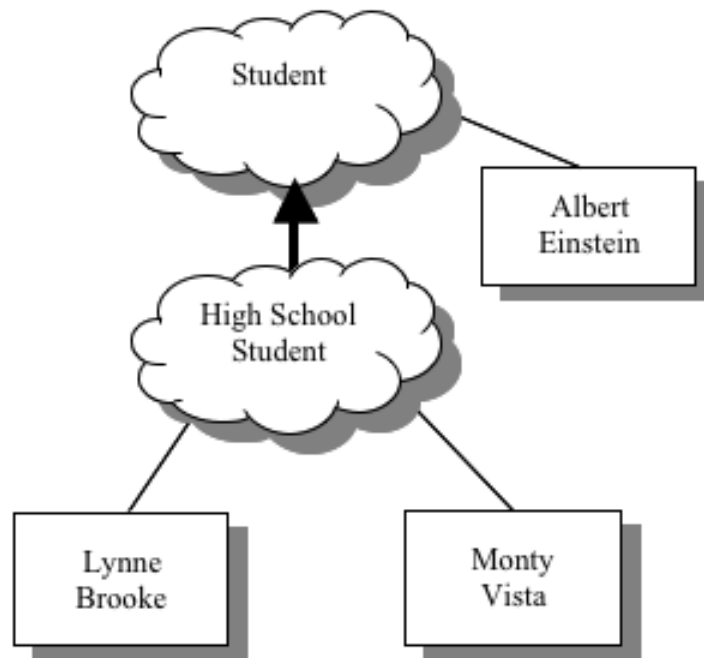


Figure 11.2 - Subclass and Superclass

6. Figure 11.2 shows a superclass and a subclass, and some objects that have been constructed from each. These objects that are shown as rectangles are actual instances of the class. In the picture, Albert Einstein, Lynne Brooke, and Monty Vista represent actual objects.

B. Class Hierarchies

page 4 of 10

1. In a hierarchy, each class has at most one superclass, but might have several subclasses. There is one class, at the top of the hierarchy that has no superclass. This is sometimes called the root of the hierarchy.

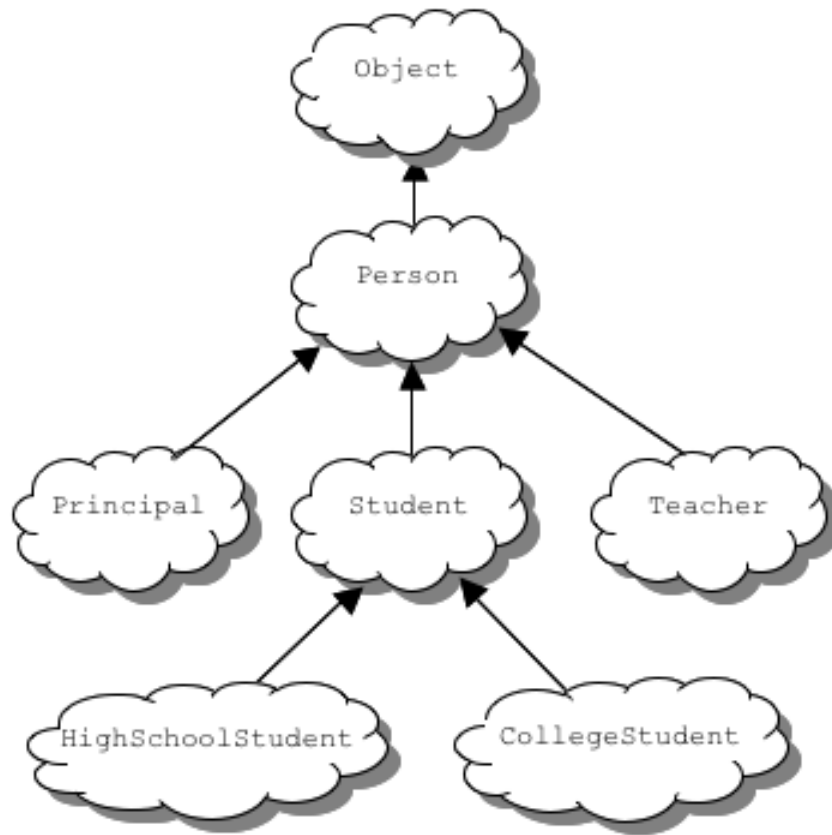


Figure 11.3 - Person Inheritance Hierarchy

Figure 11.3 shows a hierarchy of classes. It shows that a Principal is a Person, a Student is a Person, and that a Teacher is a Person. It also shows that both HighSchoolStudent and CollegeStudent are types of Student.

2. In our example, the class Person is the base class and the classes Principal, Student, Teacher, HighSchoolStudent, and CollegeStudent are derived classes.
3. In Java, the syntax for deriving a child class from a parent class is:

```
class subclass extends superclass{  
// new characteristics of the subclass go here  
}
```

4. Several classes are often subclasses of the same class. A subclass may in turn become a parent class for a new subclass. This means that inheritance can extend over several "generations" of classes. This is shown in Figure 11.3, where class HighSchoolStudent is a subclass of class Student, which is itself a subclass of the Person class. In this case, class HighSchoolStudent is considered to be a subclass of the Person class, even though it is not a direct subclass.
5. In Java, every class that does not specifically extend another class is a subclass of the class object. For example, in Figure 11.3, the Person class extends the class object. The class object has a small number of methods that make sense for all objects, such as the toString method, but the class object's implementations of these methods are

not very useful and the implementations usually get redefined in classes lower in the hierarchy.

C. Using Inheritance

page 5 of 10

1. The following program uses a class `Person` to represent people you might find at a school. The `Person` class has basic information in it, such as name, age and gender. An additional class, `Student`, is created that is similar to `Person`, but has the Id number and grade point average of the student.

```
public class Person{
    private String myName;    // name of the person
    private int myAge;        // person's age
    private String myGender;  // "M" for male, "F" for female

    // constructor
    public Person(String name, int age, String gender){
        myName = name;
        myAge = age;
        myGender = gender;
    }

    public String getName(){
        return myName;
    }

    public int getAge(){
        return myAge;
    }

    public String getGender(){
        return myGender;
    }

    public void setName(String name){
        myName = name;
    }

    public void setAge(int age){
        myAge = age;
    }

    public void setGender(String gender){
        myGender = gender;
    }
}
```

```

    public String toString(){
        return myName + ", age: " + myAge + ", gender: "
            + myGender;
    }
}

//-----End of Person Class-----//

public class Student extends Person{
    private String myIdNum; // Student Id Number
    private double myGPA; // grade point average

    // constructor
    public Student(String name, int age, String gender,
                    String idNum, double gpa){
        // use the super class' constructor
        super(name, age, gender);

        // initialize what's new to Student
        myIdNum = idNum;
        myGPA = gpa;
    }

    public String getIdNum(){
        return myIdNum;
    }

    public double getGPA(){
        return myGPA;
    }

    public void setIdNum(String idNum){
        myIdNum = idNum;
    }

    public void setGPA(double gpa){
        myGPA = gpa;
    }
}

//-----End of Student Class-----//

public class HighSchool{
    public static void main (String args[]){
        Person bob = new Person("Coach Bob", 27, "M");
        Student lynne = new Student("Lynne Brooke", 16, "F",
                                    "HS95129", 3.5);

        System.out.println(bob);
        System.out.println(lynne);
        // The previous two lines could have been written as:
        // System.out.println(bob.toString());
        // System.out.println(lynne.toString());
    }
}

```

}

2. The `Student` class is a derived class (subclass) of `Person`. An object of type `Student` contains `myIdNum` and `myGPA`, which are defined in `Student`. It also has indirect access to the private variables `myName`, `myAge`, and `myGender` from `Person` through the methods `getName()`, `getAge()`, `getGender()`, `setName()`, `setAge()`, and `setGender()` that it inherits from `Person`.
3. The constructor for the `Student` class initializes the instance data of `Student` objects and uses the `Person` class's constructor to initialize the data of the `Person` superclass. The constructor for the `Student` class looks like this:

```
// constructor
public Student(String name, int age, String gender,
               String idNum, double gpa){
    // use the super class's constructor
    super(name, age, gender);

    // initialize what's new to Student
    myIdNum = idNum;
    myGPA = gpa;
}
```

The statement `super(name, age, gender)` invokes the `Person` class's constructor to initialize the inherited data in the superclass. The next two statements initialize the members that only `Student` has. Note that when `super` is used in a constructor, it must be the *first* statement.

4. So far, we have only seen the `public` (class members that can be accessed outside the class) and `private` (class members that are inaccessible from outside the class) access modifiers. There is a third access modifier that can be applied to an instance variable or method. If it is declared to be `protected`, then it can be used in the class in which it is defined and in any subclass of that class. This declaration is less restrictive than `private` and more restrictive than `public`. The A.P. Java subset allows the use of `protected` with methods but discourages its use for instance variables. It is preferred that all instance variables are `private`. Indirect access from subclasses should be done with `public` "getter" and "setter" methods. While `protected` members are available to provide a foundation for the subclasses to build on, they are still invisible to the public at large.

1. A derived class can *override* a method from its base class by defining a replacement method with the same signature. For example, in our `Student` subclass, the `toString()` method contained in the `Person` superclass does not reference the new variables that have been added to objects of type `Student`, so nothing new is printed out. We need a new `toString()` method in the class `Student`:

```
// overrides the toString method in the parent class
public String toString(){
    return getName() + ", age: " + getAge() + ", gender: "
        + getGender() + ", student id: " + myIdNum
        + ", gpa: " + myGPA;
}
```

A more efficient alternative is to use **super** to invoke the `toString()` method from the parent class while adding information unique to the `Student` subclass:

```
public String toString(){
    return super.toString() +
        ", student id: " + myIdNum + ", gpa: " + myGPA;
}
```

2. Even though the base class has a `toString()` method, the new definition of `toString()` in the derived class will override the base class's version. The base class has its method, and the derived class has its own method with the same name. With the change in the `Student` class the following program will print out the full information for both items.

```
Person bob = new Person("Coach Bob", 27, "M");
Student lynne = new Student("Lynne Brooke", 16, "F",
    "HS95129", 3.5);

System.out.println(bob.toString());
System.out.println(lynne.toString());
```

The output to this block of code is:

```
Coach Bob, age: 27, gender: M
Lynne Brooke, age: 16, gender: F, student id: HS95129, gpa: 3.5
```

The line `bob.toString()` calls the `toString()` method defined in `Person`, and the line `lynne.toString()` calls the `toString()` method defined in `Student`.

1. In Java, an interface is a mechanism that unrelated objects use to interact with each other. Like a protocol, an interface specifies agreed-on behaviors and/or attributes.
2. The `Person` class and its class hierarchy define the attributes and behaviors of a person. But a person can interact with the world in other ways. For example, an employment program could manage a person at a school. An employment program isn't concerned with the kinds of items it handles as long as each item provides certain information, such as salary and employee ID. This interaction is enforced as a protocol of method definitions contained within an interface. The `Employable` interface would define, but not implement, methods that set and get the salary, assign an ID number, and so on.

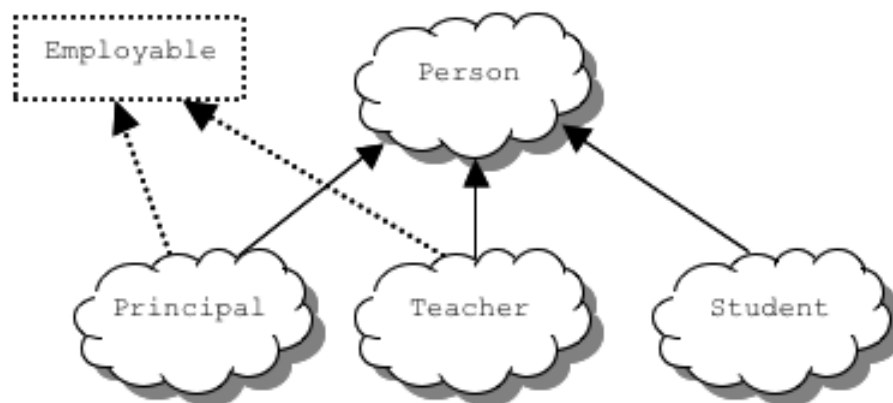


Figure 11.4 - Employable Interface

3. To work in the employment program, the `Teacher` class must agree to this protocol by *implementing* the interface. To implement an interface, a class must implement all of the methods and attributes defined in the interface. In our example, the shared methods of the `Employable` interface would be implemented in the `Teacher` class.
4. In Java, an **interface** consists of a set of methods and/or methods, without any associated implementations. Here is an example of Java interface that defines the behaviors of “employability” described earlier:

```
public interface Employable{
    public double getSalary();
    public String getEmployeeID();

    public void setSalary(double salary);
    public void setEmployeeID(String id);
}
```

A class *implements* an *interface* by defining all the attributes and methods defined in the *interface*. `implements` is a reserved word. For example:

```

public class Teacher implements Employable{
    ...
    public double getSalary() { return mySalary; }
    public int getEmployeeID() { return myEmployeeID; }

    public void setSalary(double salary) { mySalary = salary; }
    public void setEmployeeID(String id) { myEmployeeID = id; }
}

```

5. A class can implement any number of interfaces. In fact, a class can both extend another class and implement one or more interfaces. So, we can have things like (assuming we have an interface named `Californian`)

```

public class Teacher extends Person implements Employable, Californian{
    ...
}

```

6. Interfaces are useful for the following:

- Declaring a common set of methods that one or more classes are required to implement
- Providing access to an object's programming interface without revealing the details of its class.
- Providing a relationship between dissimilar classes without imposing an unnatural class relationship.

7. You are not likely to need to write your own interfaces until you get to the point of writing fairly complex programs. However, there are a few interfaces that are used in important ways in Java's standard packages. You'll learn about some of these standard interfaces in future lessons.

Summary/Review

page 8 of 10

Inheritance represents the "is a" relationship between types of objects. In practice it may be used to simplify the creation of a new class. It is the primary tool for reusing your own and standard library classes. Inheritance allows a programmer to derive a new class (called a derived class or a subclass) from another class (called a base class or superclass). A derived class inherits all the data fields and methods (but not constructors) from the base class and can add its own methods or redefine some of the methods of the base class. With the size and complexity of modern programs, reusing code is the only way to write successful programs in a reasonable amount of time.

Chapter 12

A12 Introduction

page 1 of 18

Solving problems on a computer very often requires a repetition of a block of code. Reading in data from a file, outputting to a file or adding numbers are situations where repetition is required. In Lesson A9, Recursion, we have already explored repeating code. However, not all iterative problems lend themselves to recursive solutions. Java provides three alternative constructs for repeating code with the `for` loop, the `while` loop, and the `do-while` loop. The `while` and `for` control structures allow us to set up a conditional loop, one that occurs for an indefinite period of time until some condition becomes false. We will also study the optional `do-while` loop and the concept of nested loops.

The key topics for this lesson are:

- A. [The while Loop](#)
- B. [Loop Boundaries](#)
- C. [Conditional Loop Strategies](#)
- D. [The for Loop](#)
- E. [Nested Loops](#)
- F. [The do-while Loop](#) (optional)
- G. [Choosing a Loop Control Structure](#)
- H. [Loop Invariants](#)

[A12 Vocabulary](#)

page 2 of 18

break	BOUNDARY
do-while	ENTRY CHECK
EXIT CHECK	for
LOOP INVARIANT	NESTED LOOP
SENTINEL	STATE
while	

A. The `while` Loop

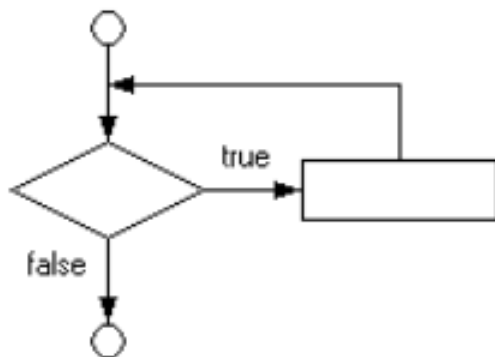
page 3 of 18

1. The general form of a `while` statement is:

```
while (expression){  
    statement;  
}
```

- a. As in the `if-else` control structure, the boolean expression must be enclosed in parentheses `()`.
 - b. The statement executed by the `while` loop can be a simple statement, or a compound statement blocked with braces `{}`.
2. If the expression is true, the statement is executed. After execution of the statement, program control returns to the top of the **`while`** construct. The statement will continue to be executed until the expression evaluates to false.
 3. The following diagram illustrates the flow of control in a `while` loop:

while structure



4. The following loop will print out the integers from 1-10.

```
int number = 1;                // initialize  
  
while (number <= 10){          // loop boundary condition  
    System.out.println(number);
```

```
    number++;                // increment/decrement
}
```

5. The above example has three key lines that need emphasis.
 - a. You must initialize the loop control variable. If you do not initialize `number`, Java produces an error message warning you that the variable may not have been initialized.
 - b. The loop boundary conditional test (`number <= 10`) is often a source of error. Make sure that you have the correct comparison (`<`, `>`, `==`, `<=`, `>=`, `!=`) and that the boundary value is correct. Programmers have to ensure that the loop is executed exactly the correct number of times. Performing the loop one too many or one too few times is called an OBOB , Off By One Bug.
 - c. There must be some type of increment/decrement or other statement so that execution of the loop eventually terminates, otherwise the program will get stuck in an infinite loop and never end!
6. It is possible for the body of a `while` loop to execute zero times. The `while` loop is an entry check loop. If the condition is false due to some initial value, the statement inside of the `while` loop will never happen. This is appropriate in some cases.

B. Loop Boundaries

page 4 of 18

1. The loop boundary is the boolean expression that evaluates as true or false. We must consider two aspects as we devise the loop boundary.
 - a. It must eventually become false, which allows the loop to exit.
 - b. It must be related to the task of the loop. When the task is done, the loop boundary must become false.
2. There are a variety of loop boundaries of which two will be discussed in this section.
3. The first is the idea of attaining a certain count or limit. The code in section A.4 above is an example of a count type of boundary.
4. Sample problem: In the margin to the left, write a program fragment that prints the even numbers 2-20. Use a `while` loop.

5. A second type of boundary construction involves the use of a sentinel value. In this category, the while loop continues until a specific value is entered as input. The loop watches out for this sentinel value, continuing to execute until this special value is input and then breaking out from the loop. You have already used the break statement for switch statements, and it has a similar use here. Once the loop encounters the break statement, all further checks of the boundary condition are ignored and code execution continues after the end of the while loop. For example, here is a loop that keeps a running total of non-zero integers, terminated by a value of zero.

```
Scanner in = new Scanner(System.in);
int total = 0;
int number;

while (true){
    System.out.print ("Enter a number (0 to quit) --> ");
    number = in.nextInt();
    if(number == 0){
        break;
    }else{
        total += number;
    }
}
System.out.println("Total = " + total);
```

Notice that because we don't know how many times we want the loop to run, we simply declare the boundary condition as always true. This means the loop will run until we tell it to stop with the break command.

6. A similar construct to the break statement is the continue statement. When a loop encounters a continue statement, every statement left to execute *in that specific iteration* is ignored. The loop will then go back to check its boundary condition like normal. Continue statements can be useful for ignoring special cases (such as if you want to ignore an entry of zero in a loop that may use that number as a divisor).

C. Conditional Loop Strategies

page 5 of 18

1. This section will present a variety of strategies that assist the novice programmer in developing correct while loops. The problem to be solved is described first.

Problem statement:

A program will read integer test scores from the keyboard until a negative value is typed in. The program will drop the lowest score from the total and print the average of the remaining scores.

2. One strategy in designing a `while` loop is to think about the following four sections of the loop: 1) initialization, 2) loop boundary, 3) contents of the loop and 4) the state of variables after the loop.
 - a. Initialization - Variables will usually need to be initialized before you get into the loop. This is especially true of `while` loops since the boundary condition is at the top of the control structure.
 - b. Loop boundary - You must construct a Boolean expression that becomes false when the problem is done. This is the most common source of error in coding a `while` loop. Be careful of off-by-one errors that cause the loop to happen one too few or one too many times.
 - c. Contents of the loop - This is where the problem is solved. The statement of the loop must also provide the opportunity to reach the loop boundary. If there is no movement toward the loop boundary, you will get stuck in an infinite loop.
 - d. State of variables after the loop - To ensure the correctness of your loop you must determine the status of key variables used in your loop. One way to do this is by tracing the code on paper.
3. We now solve the problem by first developing pseudocode.

Pseudocode:

```
initialize total and count to 0
initialize smallest to Integer.MAX_VALUE
get first score
while score is not a negative value
    increment total
    increment count
    change smallest if necessary
    get next score
subtract smallest from total
calculate average
```

4. And now it is easy to develop a working loop from this concise and easy to read pseudocode.
5. Tracing code is best done in a chart or table format. It keeps your data organized

better than marking values all over the page. We now trace the following sample data input.

65 23 81 17 45 -1

<i>score</i>	<i>score >= 0</i>	<i>total</i>	<i>count</i>	<i>smallest</i>
undefined	undefined	0	0	INT_MAX
65	true	65	1	65
23	true	88	2	23
81	true	169	3	23
17	true	186	4	17
45	true	231	5	17
-1	false			

When the loop is terminated, the three key variables (*total*, *score*, and *smallest*) contain the correct answers.

D. The for Loop

page 6 of 18

1. The for loop has the same effect as a while loop, but uses a different format. The general form of a for loop is:

```
for (statement1; expression2; statement3){  
    statement4;  
}
```

The **for** loop is typically set up as follows.

statement1 initializes the loop variable

expression2 is a **boolean** expression

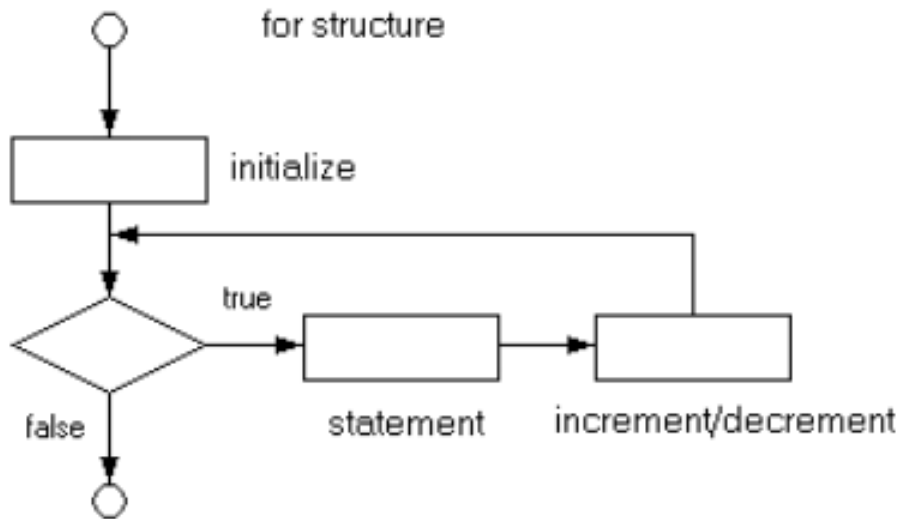
statement3 alters the key value, usually via an increment/decrement statement

statement4 is the task to be done during each iteration

2. Here is an example of a **for** loop, used to print the integers 1-10.

```
for (int loop = 1; loop <= 10; loop++){  
    System.out.print(loop);  
}
```

3. The flow of control in a for loop is illustrated below:



Notice that after the statement is executed, control passes to the increment/decrement statement, and then back to the Boolean condition.

4. The following is the equivalent **while** loop:

```
int loop = 1;
while (loop <= 10){
    System.out.print( loop);
    loop++;
}
```

5. A **for** loop is appropriate when the initialization value and number of iterations is known in advance. The above example of printing 10 numbers is best solved with a **for** loop because the number of iterations of the loop is well defined.
6. Constructing a **for** loop is easier than a **while** loop because the key structural parts of a loop (initialization, loop boundary, and increment/decrement statement) are contained in one line. It is also easier to visually check the correctness of a **for** loop because it is so compact.
7. A while loop is more appropriate when the boundary condition is tied to some input or changing value inside of the loop.
8. Here is an interesting application of a for loop to print the alphabet:

```
char letter;

for (letter = 'A'; letter <= 'Z'; letter++){
    System.out.print(letter);
}
```

The increment statement `letter++` will add one to the ASCII value of `letter`.

9. A simple, but time-consuming error to find and fix is the accidental use of a null statement.

```
for (loop = 1; loop <= 10; loop++); // note __;__  
    System.out.print(loop);
```

The semicolon placed at the end of the first line causes the `for` loop to do "nothing" 10 times. The output statement will only happen once after the `for` loop has done the null statement 10 times. The null statement can be used as a valid statement in control structures. Make sure you pay attention to your enclosing `{}`.

10. There are two basic options for the variable used in the `for` loop. The variable can either be declared beforehand and therefore initially have a value set at another point in the program, or it can be declared and initialized within the `for` loop itself. Consider the following two `for` loops:

```
int number = 1;  
  
for(; number <= 10; number++){  
    System.out.println(number);  
}  
  
for(int a = 1; a <= 10; a++){  
    System.out.println(a);  
}
```

Notice how the first statement of the first `for` loop is blank. This is because the `int` variable `number` has already been declared and initialized. Blank statements within the `for` loop are allowed. Now, both `for` loops appear to do the exact same thing, printing out all the numbers from 1 to 10. However, there are several key differences. In the first example, `number` may be changed to any number desired during the run of the program. Instead of setting it to one, a programmer could set it to the value of a function such as `int number = getValidNumber()`. This gives additional power to the programmer. Another difference between the two loops is the scope of the variables `number` and `a`. Because `number` was declared outside of the `for` loop, the value of `number` may be used after the `for` loop. However, `a` was declared within the `for` loop and thus will not be usable past the end bracket of the `for` loop.

1. To nest loops means to place one loop inside of another loop. The statement of the outer loop will be another inner loop.
2. The following example will print a rectangular grid of stars with 4 rows and 8 columns.

```
for (int row = 1; row <= 4; row++){
    for (int col=1; col <= 8; col++){
        System.out.print("*");
    }
    System.out.println( );
}
```

Run Output:

```
*****
*****
*****
*****
```

3. For each occurrence of the outer `row` loop, the inner `col` loop will print 8 stars, terminated by the newline character.
4. The action of nested loops can be analyzed using a chart:

row	col
1	1 to 8
2	1 to 8
3	1 to 8
4	1 to 8

5. Suppose we wanted to write a method that prints out the following 7-line pattern of stars:

```
*****
 *****
  *****
   *****
    *****
     *****
      *****
```

6. Here is an analysis of the problem, line-by-line.

Line #	# spaces	# stars
---------------	-----------------	----------------

Created by mu

1	0	7
2	1	6
3	2	5
...		
7	6	1
L	L - 1	N - L + 1

For a picture of n lines, each line L will have $(L-1)$ spaces and $(N-L+1)$ stars.

7. Here is a pseudocode version of the method.

A method to print a pattern of stars:

Print n lines of stars, each Line L consists of
 $(L-1)$ spaces
 $(N-L+1)$ stars
a line feed

8. Here is the code version of the method.

```
void picture (int n){
    int line, spaces, stars, loop;

    for (line = 1; line <= n; line++){
        spaces = line - 1;
        for (loop = 1; loop <= spaces; loop++){
            System.out.print (" "); // print a blank space
        }
        stars = n - line + 1;
        for (loop = 1; loop <= stars; loop++){
            System.out.print ("*");
        }
        System.out.println();
    }
}
```

F. The do-while Loop (Optional)

page 8 of 18

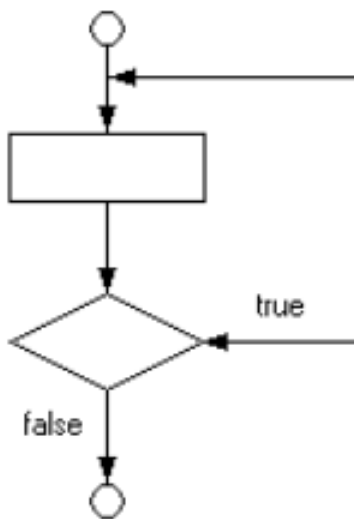
1. There are conditional looping situations where it is desirable to have the loop execute at least once, and then evaluate an exit expression at the end of the loop.

2. The **do-while** loop allows you to do a statement first, and then evaluate an exit condition. The **do-while** loop complements the **while** loop that evaluates the exit expression at the top of the loop.
3. The general form of a **do-while** loop is:

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

4. The flow of control for a **do-while** loop is illustrated:

do-while structure



5. The following fragment of code will keep a running total of integers, terminated by a sentinel zero value.

```
int number, total = 0;  
do{  
    System.out.print("Enter an integer (0 to quit) --> ");  
    number = in.readInt();  
    total += number;  
}while (number != 0);
```

In contrast to the **while** loop version, the **do-while** has the advantage of using only one input statement inside of the loop. Because the Boolean condition is at the bottom, you must pass through the main body of a **do-while** loop at least once.

6. The same strategies used to develop **while** loops apply to **do-while** loops. Make sure you think about the following four sections of the loop: initialization, loop boundary, contents of the loop, and the state of variables after the loop.

1. If you know how many times a loop is to occur, use a **for** loop. Problems that require execution of a pre-determined number of loops should be solved with a **for** statement.

See [Handout A12.1](#), *Programming Pointers*

2. The key difference between a **while** and **do-while** loop is the location of the boundary condition. In a **while** loop, the boundary condition is located at the top of the loop. Potentially, the statements within a **while** loop could happen zero times. If it is possible for the algorithm to occur zero times, use a **while** loop.
3. Because a **do-while** loop has its boundary condition at the bottom of the loop, the loop body must occur at least once. If the nature of the problem being solved requires at least one pass through the loop, use a **do-while** loop.


1. A loop invariant is an assertion about the loop that is relevant to the purpose of the loop. It is a precise statement, in terms of the loop variables, of what is true before and after each iteration of the loop.
2. Loop invariants are used to reason about programs formally and to prove their correctness without tracing all the iterations through a loop. If you can establish that an assertion is true the first time the loop is evaluated as well as after each iteration of the loop body, then your assertion is a loop invariant.
3. Consider the following code segment. Note that `count!` means the factorial of `count`.

```
int factorial (int num){  
    int product = 1;  
    int count = 0;  
  
    while (count < num){ // invariant: product == count!
```

```

        count += 1;
        product *= count;
    }
    return product;
}

```

Each time that the loop test is evaluated, the value of the variable `product` is always equal to $(\text{count})!$. Since $0! = 1$ (by definition), this is true the first time the loop test is evaluated as well as after each iteration of the loop body. Since `product == count!` is true each time the loop test is evaluated, it is the loop invariant.  the truth of the statement does not vary or change. Loop invariants are useful in reasoning about the correctness of programs that use loops. Since `product == count!` is an invariant, and `product` is returned, we can reason that the factorial method calculates the correct value.

LAB ASSIGNMENT A12.4

page 15 of 18

Grades

Background:

Suppose that a high school district decided on the following academic standards for a student's eligibility to participate in extracurricular activities, such as athletics, music, or drama, etc.:

1. No F's.
2. Minimum 2.0 grade point average (gpa)
Note: A = 4.0, B = 3.0, C = 2.0, D = 1.0, F = 0
3. Enrollment in a minimum of four academic classes

Assignment:

1. Write a program that accepts the letter grades for a student, calculates the student's gpa, and prints it out, along with one of the following five messages:

Eligible

Ineligible, taking less than 4 classes

Ineligible, gpa below 2.0

Ineligible, gpa above 2.0 but has F grade (note: $\text{gpa} \geq 2.0$)

Ineligible, gpa below 2.0 and has F grade

2. Your program must use an appropriate sequence of nested if-else statements to print out the appropriate message.
3. The message "Ineligible, taking less than 4 classes" has priority over the other 3 ineligible cases.
4. The class will not ask the user for how many grades are in a student's report card. The program will continue to read grades until a non-grade character is input. At this point, some type of loop will cease and the program prints the GPA value and the eligibility message.
5. Example of run output: GPA = 3.75 Eligible
6. You do not have to print out any of the individual grades.
7. Your program should allow input of grades in either upper or lower case.

Instructions:

1. Use these 6 sample report cards (with a grade for each academic class) as inputs for your run outputs:

B B C B F	C B D D D C
C D C	A A B A A B A
A B A	D C F F D

LAB ASSIGNMENT A12.5

page 16 of 18

Payments

Background:

Borrowing money for expensive items has become a way of life for most Americans. To illustrate the high cost of borrowing and how such loans work, you will be writing a class to calculate the following monthly analysis of a loan.

Month	Principal	Loan Amt.	Interest (at 1%/month)	Payment	New Balance
-------	-----------	-----------	---------------------------	---------	-------------

1	10000.00	100.00	300.00	9800.00
2	9800.00	98.00	300.00	9598.00
3	9598.00	95.98	300.00	9393.98
4	9393.98	93.94	300.00	9187.92

and many months later ...

39	809.46	8.09	300.00	517.55
40	517.55	5.18	300.00	222.73

Total: 2222.73

The loan analysis above started with the following information:

Principal (amount borrowed) = 10000.00

Annual Interest Rate = 12.0 %

Monthly Payment = 300.00

The monthly interest rate is found by dividing the annual rate among 12 months. For the above example the monthly rate is 1.0 %. The last three values of each line are calculated as follows:

Interest = Principal * Monthly Interest Rate

Payment = amount set at beginning of problem

New Balance = Principal + Interest - Payment

The new balance becomes the starting principal amount for the next month. As you can see, progress toward decreasing the principal is slow at the beginning of the loan.

Assignment:

Write a class to represent a loan as described above using the five-column format. Your class must accomplish the following:

1. Data input: The class should ask for the appropriate starting information.
2. Printing of analysis: The class must print the month-by-month analysis until the remaining principal is less than the monthly payment. At the bottom of the analysis you must print the total interest paid to the lending institution.

Instructions:

1. Here are some sample data sets:

Principal = 12000
Annual Interest Rate = 8.80
Monthly Payment = 500.00

Principal = 10000
Annual Interest Rate = 12.0
Monthly Payment = 300.00

Extending the Lab:

1. To vary the assignment, we now want to save some money and earn interest. Suppose we wish to study the effect of time and compounding interest on investments. Add an option to ask the user for:

Starting Principal to invest
Annual Rate of Return (5%, 10%, etc)
Monthly Addition to the Principal
Number of Months to Iterate

2. The printout will be similar except the column called “payment” will be changed to investment. You should still calculate and print out the total interest and final balance.
3. This lab exercise should encourage you to start investing early in life.

LAB ASSIGNMENT A12.6

page 17 of 18

ParallelLines

Background:

In this lab, you will practice using nested `for` loops to recreate this image using the `DrawingTool` class:

To do this efficiently, there are a few suggestions to consider. First, there are eight rows each containing seven filled boxes. This suggests a nested `for` loop, like this:

```
for (int row = 0; row < 8; row++){  
    // calculate the start of the row of squares
```

```

    // calculate and add a horizontal offset

    for (int col = 0; col < 7; col++){
        // draw the square
        // calculate and position for the next square
    }

    // calculate the location and draw the line
}

```

Calculating the square position will take some work. You might want to make a quick sketch, with coordinates, to save frustration. Hint: there will be a negative x offset and a positive y offset to have the image start in the upper left corner as shown.

The DrawingTool class includes a method for drawing filled rectangles:

```
fillRect(double width, double height)
```

For instance, with a DrawingTool called pen, the call `pen.fillRect(40, 40)` would create a 40x40 filled square centered about the current drawing position.

Assignment:

1. Using the information given in the Background section above, write a class using nested for loops to display the image shown above.

LAB ASSIGNMENT A12.7

page 18 of 18

GameLand

Background:

This exercise is based on the American Computer Science League - Intermediate Division, Contest #1, 1989-90.

1. The board game GameLand is a very simple one that you will simulate on the computer.
2. Two players begin the game on a square labeled as "START." The goal of the game is to be the first player to reach the square labeled "FINISH." There are 100 squares between START and FINISH.
3. The two players take turns rolling two six-sided dice. A roll of 2 or 12 means that the

player loses that turn and cannot move. Getting a roll of 7 means that the player moves backwards 7 spaces (but not beyond START). On all other rolls, the players must move forward an amount equal to the number of the roll.

4. If one player lands on a square occupied by the other player, the player originally on that square gets bumped back to the START square.
5. The game ends when one player wins by landing on or beyond the FINISH square.

Assignment:

1. Write a program that simulates the activities of GameLand.
2. Make sure your two dice rolls are realistic. Remember, the results are not simply 1 through 12.

Summary/Review

page 11 of 18

This lesson provides both syntax and strategies needed to build correct **while** and **for** loops. The terminology of loop construction will give us tools to build and debug conditional loops. We can use terms such as "off-by-one" errors or "failure to maintain state." This is a critical topic, one that takes much time and practice to master. Learning to translate thoughts into computer algorithms is one of the more challenging aspects of programming. We solve repetitive and selection problems constantly without thinking about the sequence of events. Using pseudocode helps translate your thinking into code.

Chapter 13

A13 Introduction

page 1 of 12

Java provides a structured approach for dealing with errors that can occur while a program is running. This approach is referred to as "exception-handling." The word "exception" is

meant to be more general than “error.” Exception-handling is used to keep a program running even though an error is encountered that would normally stop the program. This lesson will explore file input and output (I/O) as an example of how exceptions are used.

The key topics for this lesson are:

- A. [Exceptions](#)
- B. [Handling Exceptions](#)
- C. [Exception Messages](#)
- D. [Throwing Exceptions](#)
- E. [Reading From a File](#)
- F. [Writing to a File](#)

[A13 Vocabulary](#)

page 2 of 12

catch

EXCEPTION

ERROR

try

A. Exceptions

page 3 of 12

1. When a Java program performs an illegal operation, a special event known as an *exception* occurs. An exception represents a problem that the compiler was unable to detect before the execution of the program. This is called a run-time error. An example of this would be dividing by zero. The compiler often cannot tell before the program runs that a denominator would be zero at some later point and therefore cannot give an error before the program is run.
2. An exception is an object that holds information about a run-time error. The programmer can choose to ignore the exception, fix the problem and continue processing, or abort the execution of the code. An *error* is when a program does not do what it was intended to do. Compile time errors occur when the code entered into the computer is not valid. Logic errors are when all the code compiles correctly but the

logic behind the code is flawed. Run-time errors happen when Java realizes during execution of the program that it cannot perform an operation.

3. Java provides a way for a program to detect that an exception has occurred and execute statements that are designed to deal with the problem. This process is called *exception handling*. If you do not deal with the exception, the program will stop execution completely and send an exception message to the console.
4. Common exceptions include:

- ArithmeticException
- NullPointerException
- ArrayIndexOutOfBoundsException
- ClassCastException
- IOException

For example, if you try to divide by zero, this causes an `ArithmeticException`. Note that in the code section below, the second `println()` statement will not execute. Once the program reaches the divide by zero, the execution will be halted completely and a message will be sent to the console:

```
int numerator = 23;
int denominator = 0;

// the following line produces an ArithmeticException
System.out.println(numerator/denominator);

System.out.println(_This text will not print_);
```

A `NullPointerException` occurs if you use a null reference where you need an object reference. For example,

```
String name = null;

// the following line produces a NullPointerException
int i = name.length();
System.out.println(_This text will not print_);
```

Since `name` has been declared to be a reference to a `String` and has the value `null`, indicating that it is not referring to any `String` at this time, an attempt to call a method within `name`, such as `name.length()`, will cause a `NullPointerException`. If you encounter this exception, look for an object that has been declared but has not been instantiated.

1. There are three ways of handling a possible exception occurrence (we say that the exception is *thrown*). In some cases, such as when there is a possibility that a divide by zero exception might occur, the programmer has the option of not dealing with the exception at all. This can be useful in situations where the programmer has already taken steps to ensure that a denominator never becomes zero. The programmer may also attempt to fix the problem or skip over the problem. To do either of these, the programmer needs to catch any exception that may be thrown. To *catch* an exception, one must anticipate where the exception might occur and enclose that code in a `try` block. The `try` block is followed by a `catch` block that catches the exception (if it occurs) and performs the desired action.

2. The general form of a try-catch statement is:

```
try{  
    try-block  
} catch (exception-type identifier){  
    catch-block  
}
```

- a. The try-block refers to a statement or series of statements that might throw an exception. If no exception is thrown, all of the statements within the try-block will be executed. Once an exception is thrown, however, all of the statements following the exception in the try-block will be skipped.
 - b. The catch-block refers to a statement or series of statements to be executed if the exception is thrown. A try block can be followed by more than one catch block. When an exception is thrown inside a try block, the first matching catch block will handle the exception.
 - c. exception-type specifies what kind of exception object the catch block should handle. This can be specific or it can be general, i.e. `IOException` or just `Exception`. `Exception` by itself will catch any type of exception that comes its way.
 - d. identifier is an arbitrary variable name used to refer to the exception-type object. Any operations done on the exception object or any methods called will use this identifier
3. The `try` and `catch` blocks work in a very similar manner to the `if-else` statement and can be placed anywhere that normal code can be placed.
4. If an exception is thrown anywhere in the `try` block which matches one of the

exception-types named in a `catch` block, then the code in the appropriate `catch` block is executed. If the `try` block executes normally, without an exception, the `catch` block is ignored.

5. Here is an example of `try` and `catch`:

```
int quotient;
int numerator = 23;
int denominator = 0;
try{
    quotient = numerator/denominator;
    System.out.println("The answer is: " + quotient);
} catch (ArithmeticException e){
    System.out.println("Error: Division by zero");
}
```

The value of `denominator` is zero so an `ArithmeticException` will be thrown whenever `numerator` is divided by `denominator`. The `catch` block will catch the exception and print an error message. The `println()` statement in the `try` block will not be executed because the exception occurs before the program reaches that line of code. Once an exception is encountered, the rest of the lines of code in the `try`-block will be ignored. If the value of `denominator` is not zero, the code in the `catch` block will be ignored and the `println()` statement will output the result of the division. Either way, the program continues executing at the next statement after the `catch` block.

C. Exception Messages

page 5 of 12

1. If a program does not handle exceptions at all, it will stop the program and produce a message that describes the exception and where it happened. This information can be used to help track down the cause of a problem.
2. The code shown below throws an `ArithmeticException` when the program tries to divide by zero. The program crashes and prints out information about the exception:

```
int numerator = 23;
int denominator = 0;

// the following line produces an ArithmeticException
System.out.println(numerator/denominator);

System.out.println("This text will not print");
```

Run Output:

```
Exception in thread "main" java.lang.ArithmeticException: / by
zero
at DivideByZero.main(DivideByZero.java:10)
```

The first line of the output tells which exception was thrown and gives some information about why it was thrown. “DivideByZero.main” indicates that the exception occurred in the main method of the DivideByZero class. In the parentheses, the specific file name and line number are given so that the programmer can find where their code went wrong (in the example above, the exception occurred on line 10 of a Java file, DivideByZero.java). This is the line where Java found a problem. The actual root cause of the problem may be a line or two ahead of line 10.

The rest of the output tells where the exception occurred and is referred to as a call stack trace. In this case, there is only one line in the *call stack trace*, but there could be several, depending on where the exception originated.

3. When exceptions are handled by a program, it is possible to obtain information about an exception by referring to the “exception object” that Java creates in response to an exception condition. Every exception object contains a String that can be accessed using the getMessage method as follows:

```
try{
    quotient = numerator/denominator;
    System.out.println("The answer is: " + quotient);
} catch (ArithmeticException e){
    System.out.println(e.getMessage());
}
```

If a divide by zero error occurs, an exception is thrown and the following message is displayed:

```
/ by zero
```

4. Printing the value returned by getMessage can be useful in situations where we are unsure of the type of error or its cause.
5. If an exception is unknown by your Java class, you may need to import the appropriate exception (just like you would import any other class you wanted your class to know about). For example, the ArithmeticException is a standard Java class and no import statement is needed for it. However, IOException is part of the java.io package and must therefore be imported before it is used.

1. There are times when it makes sense for a program to deliberately throw an exception. This is the case when the program discovers some sort of exception or error condition, but there is no reasonable way to handle the error at the point where the problem is discovered. The program can throw an exception in the hope that some other part of the program will catch and handle the exception.
2. To throw an exception use a **throw** statement. This is usually done with an if statement. The syntax of the throw statement is:

```
throw exception-object;
```

3. For example, the following statement throws an `ArithmeticException`:

```
if(number == 0){  
    throw new ArithmeticException("Division by zero");  
}
```

The exception object is created with the new operator right in the `throw` statement. Exception classes in Java have a default constructor that takes no arguments and a constructor that takes a single `String` argument. If provided, this `String` appears in the exception message when the exception occurs.

1. Reading textual data from a file is very similar in many ways to reading input from the keyboard. The `Scanner` class and all of its methods remain the same. However, you must also import the classes `java.io.File` and `java.io.IOException`. Also, the way in which you use the `Scanner` class constructor changes.
2. The `java.io.File` is a holder class that can take a `String` representing the path to a file on your computer. Creating a `Scanner` object that reads from a file is as simple as:

```
Scanner in = new Scanner(new File("test.txt"));  
File f = new File("C:\\MyDocuments\\Java\\tester.txt");  
Scanner in2 = new Scanner(f);
```

The first line assumes that there is a file named "test.txt" in the directory or folder where you run your java files. The second line shows an example of creating the File object in its own line of code. It also shows the File being created with the full path to the file. This would be used if your text file was not in the same directory as your Java files. But what happens if the file that you are looking for doesn't exist or has some other status that prevents it from being read? This causes an exception to be thrown!

3. Scanner will take no responsibility in handling the exception, so every time that you want to use Scanner with a file, you will have to use a try-catch block. A full example is shown below:

```
Scanner in;  
try{  
    in = new Scanner(new File("test.txt"));  
    String test = in.nextLine();  
    System.out.println(test);  
}catch(IOException i){  
    System.out.println("Error: " + i.getMessage());  
}
```

4. When reading a large amount of data from a file, it is often useful to know whether there is any more data in the file to read. Reading from a file which has no more data will give you a `NoSuchElementException` and stop your program. The Scanner class has several methods for determining if there is any more data in the file to be read: `hasNext()`, `hasNextDouble()`, and `hasNextInt()` will be the methods most useful for you. If there is anything still to come, `hasNext()` will return true, while `hasNextDouble()` will return true only if a valid double is next and `hasNextInt()` will return true only if an int value is next. Using a simple while loop, you can easily read in data until the end of a file.

```
while(in.hasNext()){  
    System.out.println(in.next());  
}
```

F. Writing to a File

page 8 of 12

1. Creating your own files is a little bit trickier than reading from them. There are two basic ways to write data to files: raw bytes and character streams. Raw byte writing is useful for items such as pictures. Character streams are used for writing plain text. This curriculum will focus only on character streams.
2. This curriculum uses the `java.io.FileWriter` class. It has two basic constructors:


```
FileWriter file = new FileWriter("test.txt");
FileWriter file2 = new FileWriter("test2.txt", true);
```

The first constructor opens a basic `FileWriter` object that points at the file "test.txt" in the same directory where the Java files are being run. When this file is first opened and written to, all of the data that was previously stored in the file will be erased. The second constructor indicates that the new data being sent to the file will be appended to the end of the file. In either case, if the file does not exist, then Java will attempt to create a new file with the indicated name and at the indicated location.

3. Writing data to the file is done by using the `FileWriter.write(String, int, int)` method. The `String` is the data that will be written to the file. The first `int` is where to start writing the data in the `String`. The second `int` indicates how many characters of the `String` to actually write. For example:

```
String one = "#Hello!!!";
FileWriter out = new FileWriter("test.txt");
out.write(one, 1, 5);
```

This will write only "Hello" to the file "test.txt."

4. Merely opening a file and writing to it is not enough to store your data in most cases. You know from personal experience that if you don't save your work in a word processor, your work will not be there the next time you start up your computer. Data must be saved. This is done with `FileWriter` by calling the `close()` method when you are done writing data. This "closes" the output stream to the file and saves your data.

```
out.write(one, 1, 5);
out.close();
```

5. What if there is some error in opening the file? That's right - an exception is thrown and it must be dealt with just like in the `Scanner` class.

```
String one = "Hello World!!!";
FileWriter out;
try{
    out = new FileWriter("test.txt");
    out.write(one, 0, one.length());
    out.close();
}catch(IOException i){
    System.out.println("Error: " + i.getMessage());
}
```

6. There is no equivalent to `println()` with the `FileWriter` class, so any newlines that you wish to create must be done with the '\n' character.
7. `FileWriter` only deals with writing `Strings` to the text files, which creates a little bit of a

problem with writing numeric data. However, we can use the shortcut learned earlier in Lesson A10, *Strings* to change our other data types to Strings.

```
String temp;
int a = 5;
temp = "" + a + "\n";
out.write(temp, 0, temp.length());
double p = 3.14;
temp = "" + p + "\n";
out.write(temp, 0, temp.length());
boolean test = true;
temp = "" + test + "\n";
out.write(temp, 0, temp.length());
```

8. Because `FileWriter` requires you to specify how many characters of the given `String` to print out, you must be careful with the values that you give it. If the `int` value that you send is bigger than the `String` itself, you will get a `StringIndexOutOfBoundsException` when the `FileWriter` object tries to access characters in the `String` which do not exist. An easy way to prevent this from ever occurring is to always create a `String` object before the `write` method is called with the data you wish to output, place that `String` in the call to `write`, and use that `String`'s `length()` method for how many characters to print.

```
String one = "Hello World!!!\n";
Out.write(one, 0, one.length());
```

Summary/Review

page 9 of 12

Exceptions provide a clean way to detect and handle unexpected situations. When a program detects an error, it throws an exception. When an exception is thrown, control is transferred to the appropriate exception handler. By defining a method that catches the exception, the programmer can write the code to handle the error. Exceptions are used heavily when dealing with File I/O because there are many situations that can turn dangerous when reading and writing data directly from a hard drive.

Chapter 14

Conditional loops often prove to be one of the most difficult control structures to work with. This lesson will give you more strategies that can be used for defining the beginning and ending conditions for loops in your programs.

The key topics for this lesson are:

- A. [Negations of Boolean Assertions](#)
- B. [Boolean Algebra and DeMorgan's Laws](#)
- C. [Application of DeMorgan's Laws](#)

[A14 Vocabulary](#)

ASSERTION

BOOLEAN ALGEBRA

BOOLEAN ASSERTIONS

DE MORGAN'S LAWS

A. Negations of Boolean Assertions

1. A Boolean assertion is simply an expression that results in a true or false answer. For example,

$a > 5$ $0 == b$ $a <= b$

are all statements that will result in a true or false answer.

2. To negate a Boolean assertion means to write the opposite of a given Boolean assertion. For example, given the following Boolean assertions noted as A, the corresponding negated statements are the result of applying the ! operator to A.

<u>A</u>	<u>!A</u>
5 == x	5 != x
x < 5	x >= 5
x >= 5	x < 5

3. Notice that negations of Boolean assertions can be used to re-write code. For example:

```
if (!(x < 5))
    // do something...
```

can be rewritten as

```
if (x >= 5)
    // do something ...
```

This is important because we understand positive statements much more easily than statements that contain one or more !s.

B. Boolean Algebra and DeMorgan's Laws

page 4 of 7

1. Boolean Algebra is a branch of mathematics devoted to the study of Boolean values and operators. Boolean Algebra consists of these fundamental operands and operators:

operands (values): **true**, **false**

operators: and (&&), or (| |), not (!)

(Note: Java has other Boolean operators, such as ^ (XOR - "exclusive or") and equivalence. This curriculum does not cover these other operators because they are not part of the AP subset.)

2. There are many identities that have been developed to use with compound Boolean expressions. Two of the more useful identities are DeMorgan's Laws, which are used to negate compound Boolean expressions.

DeMorgan's Laws:

$!(A \parallel B) \rightarrow !A \&\& !B$

$!(A \&\& B) \rightarrow !A \parallel !B$

The symbols A and B represent the Boolean values, **true** or **false**.

3. Here is the truth table that proves the first DeMorgan's Law.

A	B		$!(A \parallel B)$	$!A$	$!B$	$!A\&\&!B$
true	true		false	false	false	false
true	false		false	false	true	false
false	true		false	true	false	false
false	false		true	true	true	true

Notice that columns with the titles $!(A \parallel B)$ and $!A \&\& !B$ result in the same answers.

4. Following is the truth table that proves the second DeMorgan's Law.

A	B		$!(A\&\&B)$	$!A$	$!B$	$!A \parallel !B$
true	true		false	false	false	false
true	false		true	false	true	true
false	true		true	true	false	true
false	false		true	true	true	true

Notice that columns with the titles $!(A \&\& B)$ and $!A \parallel !B$ result in the same answers.

5. Here is a good way to think about both of DeMorgan's Laws. Notice that it is similar to the distributive postulate in mathematics. The not operator is distributed through both terms inside of the parentheses, except that the operator switches from *and* to *or*, or vice versa.

$!(A \&\& B) \rightarrow !A \parallel !B$

$!(A \parallel B) \rightarrow !A \&\& !B$

1. The casino game of craps involves rolling a pair of dice. The rules of the game are as follows. (Refer to Lesson A6 if you need to review the Random class)

- If you roll a 7 or 11 on the first roll, you win.
- If you roll a 2, 3, or 12 on the first roll, you lose.
- Otherwise, rolling a 4, 5, 6, 8, 9, or 10 establishes what is called the point value.
- If you roll the point value before you roll a 7, you win. If you roll a 7 before you match the point value, you lose.
- After the point value has been matched, or a 7 terminates the game, play resumes from the top.

2. The following sequences of dice rolls give these results.

7	player wins
4 5 3 7	player loses
8 6 2 8	player wins
3	player loses

3. The rules of the game are set so that the house (casino) always wins a higher percentage of the games. Based on probability calculations, the actual winning percentage of a player is 49.29%.
4. A complete program, *Craps.java*, is provided in Handout A14.1. The application of DeMorgan's Laws occurs in the `getPoint()` method. The `do-while` loop has a compound exit condition.

See [Handout A14.1, Craps.java](#)

```
do{
    sum = rollDice();
}
while ((sum != point) && (sum != 7));
```

5. When developing a conditional loop, it is very helpful to think about what assertions are true when the loop will be finished. In other words, when the loop is done, what will be true?
6. When the loop in `getPoint` is done, one of two things will be true:
- a. the point will be matched (`sum == point`).

b. or a seven has been rolled.

These two statements can be combined into one summary assertion statement:

```
((sum == point) || (sum == 7))
```

7. The loop assertion states what will be true when the loop is done. Once you have established the loop assertion, writing the boundary condition involves a simple negation of the loop assertion.
8. Taking the assertion developed in Section 6 above, the negation of the assertion follows.

```
!((sum == point) || (sum == 7))
```

9. Applying DeMorgan's law results in

```
(!(sum == point)) && (!(sum == 7))
```

Rewriting each half of the expression gives

```
(sum != point) && (sum != 7)
```

10. Looking at the first half of the developing boundary condition, the statement `(sum != point)` means that we have not yet matched the point. In other words, we haven't won yet.
11. The second half of the boundary condition `(value != 7)` means we have not yet "crapped" out (i.e., rolled a 7). In other words, we also haven't lost yet, so we must keep rolling the dice.
12. You may use the equivalent Boolean expressions in Sections 8 and 9 interchangeably. Choose the one that you think makes your code easier to read.

Summary/Review

page 6 of 7

Conditional loops are some of the hardest pieces of code to write correctly. The goal of this lesson is to have a structured approach for the construction of conditional loops. Thinking in the positive sense is typically easier. Determine what will be true when the loop is done. Negate this condition to stay in the loop.

Chapter 15

A15 Introduction

page 1 of 11

It is very common for a program to manipulate data that is kept in a list. Lists are a fundamental feature of Java and most programming languages. Because lists are so useful, the Java Development Kit includes the `ArrayList` class. The `ArrayList` class provides the classic operations for a list.

The key topics for this lesson are:

- A. [ArrayList Implementation of a List](#)
- B. [The ArrayList Class](#)
- C. [Object Casts](#)
- D. [The Wrapper Classes](#)
- E. [Iterator](#)

[A15 Vocabulary](#)

page 2 of 11

ABSTRACT DATA TYPE	<code>ArrayList</code>
CAST	for each LOOP
LIST	WRAPPER

A. ArrayList Implementation of a List

page 3 of 11

1. A data structure combines data organization with methods of accessing and manipulating the data. For example, an `ArrayList` is a data structure for storing a list of elements and provides methods to find, insert, and remove an element. At a very abstract level, we can describe a general “list” object. A list contains a number of elements arranged in sequence. We can find a target value in a list, add elements to the list, remove elements from the list and process each element of the list.
2. An abstract description of a data structure, with the emphasis on its properties, functionality, and use, rather than on a particular implementation, is referred to as an *Abstract Data Type* (ADT). An ADT defines methods for handling an abstract data organization without the details of implementation.
3. A “list” ADT, for example, may be described as follows:

Data organization:

- Contains a number of data elements arranged in a linear sequence

Methods:

- Create an empty List
- Append an element to List
- Remove the i-th element from List
- Obtain the value of the i-th element
- Traverse List (process or print out all elements in sequence, visiting each element once)

4. An `ArrayList` object contains an array of object references plus many methods for managing that array. The most convenient feature of an `ArrayList` is that you can keep adding elements to it no matter what size it was originally. The size of the `ArrayList` will automatically increase and no information will be lost.

B. The `ArrayList` Class

page 4 of 11

1. a. To declare a reference variable for an `ArrayList`, do this:

```
// myArrayList is a reference to  
// a future ArrayList object  
ArrayList <ClassName> myArrayList;
```

An `ArrayList` is an array of references to objects of type `className`, where `className` can be any class defined by you or Java.

b. To declare a variable and to construct an ArrayList with an unspecified initial capacity, do this:

```
// myArrayList is a reference to an ArrayList
// object. The Java system picks the initial
// capacity.
ArrayList <ClassName> myArrayList =
    new ArrayList <ClassName> ();
```

This may not be very efficient. If you have an idea of what size ArrayList you need, start your ArrayList with that capacity.

c. To declare a variable and to construct an ArrayList with an initial capacity of 15, do this:

```
// myArrayList is a reference to an ArrayList
// object with an initial capacity of 15 elements.
ArrayList <ClassName> myArrayList =
    new ArrayList <ClassName> (15);
```

2. One way of accessing the elements of an ArrayList is by using an integer index. The index is an integer value that starts at 0 and goes to size()-1.

To access the object at a particular index, use:

```
// Returns the value of the element at index
Object get(int index);

System.out.println(myArrayList.get(i));
```

3. To add an element to the end of an ArrayList, use:

```
// add a reference to an Object to the end of the
// ArrayList, increasing its size by one
boolean add(Object obj);
```

Program Example 15 - 1:

```
import java.util.ArrayList;

ArrayList <String> names = new ArrayList <String> (10);

names.add("Cary");
names.add("Chris");
names.add("Sandy");
names.add("Elaine");

// remove the last element from the list
```

```
String lastOne = names.remove(names.size()-1);
System.out.println("removed: " + lastOne);
names.add(2, "Alyce"); // add a name at index 2

for (int j = 0; j < names.size(); j++)
    System.out.println(j + ": " + names.get(j));
```

Run Output:

```
removed: Elaine
0: Cary
1: Chris
2: Alyce
3: Sandy
```

4. A shorthand way to iterate through the collection is provided by a “for each” loop. A for each loop starts you at the beginning of the collection and proceeds through all of the elements. It will not allow you to skip elements, add elements or remove elements.

An example of using it on the collection created in the previous section is

```
for(String n : names){
    System.out.println(n);
}
```

5. The `add()` method adds to the end of an `ArrayList`. To set the data at a particular index, use:

```
// replaces the element at index with
// objectReference
Object set(int index, Object obj)
```

The `index` should be within 0 to `size()-1`. The data previously at `index` is replaced with `obj`. The element previously at the specified position is returned.

6. Removing an element from a list: The `ArrayList` class has a method that will remove an element from the list without leaving a hole in place of the deleted element:

```
// Removes the element at index from the list and
// returns its old value; decrements the indices of
// the subsequent elements by 1
Object remove(int index);
```

The element at location `index` will be eliminated. Elements at locations `index+1`, `index+2`, ..., `size()-1` will each be moved down one to fill in the gap.

7. Inserting an element into an `ArrayList` at a particular position: When an element is inserted at `index`, the element previously at `index` is moved up to `index+1`, and so on until the element previously at `size()-1` is moved up to `size()`. The size of the

ArrayList has now increased by one, and the capacity can be increased again if necessary.

```
// Inserts obj before the i-th element; increments
// the indices of the subsequent elements by 1
void add(int index, Object obj);
```

Inserting is different from setting an element. When `set(index, obj)` is used, the object reference previously at `index` is replaced by the new `obj`. No other elements are affected, and the size does not change.

8. Whether you are adding at the beginning, middle or end, remember that you are adding an object and must instantiate that object somewhere. Strings hide this fact.

```
names.add(2, "Alyce");
```

This statement actually creates a `String` object with the value `Alyce`.

If we are using any other object, we must instantiate the object. If we have an `ArrayList` `drawList` of `DrawingTools`, we could add a `DrawingTool` in the following way.

```
drawList.add(new DrawingTool());
```

C. Object Casts

page 5 of 11

1. Java compilers before J2SE 1.5 (codename: Tiger) did not support the typing of an `ArrayList` as shown above. This new feature is called *generics*. It is a safe way to deal with `ArrayLists`. You declare what kind of objects you are going to put in the `ArrayList`. Java will only allow you to put that type of object in, and it knows the type of object coming out. In previous versions of Java, you had to tell the compiler what kind of object you were putting in and taking out.
2. For example, consider the following:

```
ArrayList aList = new ArrayList();
aList.add("Chris");
String nameString = aList.get(0); // SYNTAX ERROR!
System.out.println("Name is " + nameString);
```

This code creates an `ArrayList` called `aList` and adds the single `String` object

"Chris" to the list. The intent of the third instruction is to assign the item "Chris" to `nameString`. The state of program execution following the `add` is that `aList` stores the single item, "Chris". Unfortunately, this code will never execute, because of a syntax error with the statement:

```
String nameString = aList.get(0); // SYNTAX ERROR!
```

The problem is a type conformance issue. The `get` method returns an `Object`, and an `Object` does not conform to a `String` (even though this particular item happens to be a `String`).

3. The erroneous instruction can be modified to work as expected by incorporating the `(String)` cast shown below.

```
String nameString = (String)aList.get(0);
```

D. The Wrapper Classes

page 6 of 11

1. Because numbers are not objects in Java, you cannot insert them directly into pre 1.5 `ArrayLists`. To store sequences of integers, floating-point numbers, or boolean values in a pre 1.5 `ArrayList`, you must use wrapper classes.
2. The classes `Integer`, `Double`, and `Boolean` wrap primitive values inside objects. These wrapper objects can be stored in `ArrayLists`.
3. The `Double` class is a typical number wrapper. There is a constructor that makes a `Double` object out of a double value:

```
Double r = new Double(8.2057);
```

Conversely, the `doubleValue` method retrieves the double value that is stored inside the `Double` object:

```
double d = r.doubleValue();
```

4. To add a primitive data type to a pre 1.5 `ArrayList`, you must first construct a wrapper object and then add the object. For example, the following code adds a floating-point number to an `ArrayList`:

```
ArrayList grades = new ArrayList();  
double testScore = 93.45;
```

```
Double wrapper = new Double(testScore);
grades.add(wrapper);
```

Or the shorthand version:

```
grades.add(new Double(93.45));
```

To retrieve the number, you need to cast the return value of the `get` method to `Double`, and then call the `doubleValue` method:

```
wrapper = (Double)grades.get(0);
testScore = wrapper.doubleValue();
```

With Java 1.5, declare your `ArrayList` to only hold `Doubles`. With a new feature called *auto-boxing* in Java 1.5, when you define an `ArrayList` to contain a particular wrapper class, you can put the corresponding primitive value directly into the `ArrayList` without having to wrap it. You can also pull the primitive directly out.

```
ArrayList grades2 <Double> = new ArrayList <Double>();
grades2.add(93.45);
System.out.println("Value is " + grades2.get(0));
```

E. Iterator

page 7 of 11

An `Iterator` is an object that is attached to the `ArrayList` and allows you to traverse the array from first to last element. Many data structures implement an `Iterator`. The `Iterator` keeps track of where it is in the list even if we are adding and removing from it. We will cover the topic of `Iterators` much more thoroughly in the AB level curriculum.

```
ArrayList <String> names = new ArrayList <String>();
names.add("George");
names.add("Nancy");
names.add("John");
```

```
Iterator iter = names.iterator();
while(iter.hasNext()){
    System.out.println(iter.next());
}
```

An `ArrayList` contains elements that are accessed using an integer index. The package `java.util` also includes a few other classes for working with objects.

Chapter 16

A16 Introduction

page 1 of 10

A single dimension array is an alternative to the `ArrayList` studied in Lesson A15.

The key topics for this lesson are:

- A. [Example of an Array](#)
- B. [Array Declarations and Memory Allocation](#)
- C. [Applications of Arrays](#)
- D. [Arrays as Parameters](#)
- E. [Arrays and Algorithms](#)

[A16 Vocabulary](#)

page 2 of 10

ALGORITHM

final

RANDOM ACCESS

TRAVERSAL

ARRAY

INDEX

SEQUENTIAL

1. The following program will introduce you to some of the syntax and usage of the array class in Java:

Code Sample 16-1:

```
int[] A = new int[6]; // an array of 6 integers
int loop;

for (loop = 0; loop < 6; loop++){
    A[loop] = loop * loop;
}
System.out.println("The contents of array A are:");
System.out.println();
for (loop = 0; loop < 6; loop++){
    System.out.print(" " + A[loop]);
}
System.out.println();
```

Run Output:

The contents of array A are:

0 1 4 9 16 25

2. An array is similar to the ArrayList. It is a linear data structure composed of adjacent memory locations, or “cells”, each holding values of the same type.

A

0	1	4	9	16	25
A[0]	A[1]	A[2]	A[3]	A[4]	A[5]

3. The variable A is an array, a group of 6 related scalar values. There are six locations in this array referenced by indexes 0 to 5. Note that indexes always start at zero, and count up by one until the last slot of the array. If there are N slots in an array, the indexes will be 0 through N-1 (for example, if N=6, the indexes are 0 through 5 or (N-1)).
4. The variable loop is used in a for loop to reference indexes 0 through 5. In this

program, the square of each index is stored in the memory location occupied by each cell of the array. The syntax for accessing a memory location of an array requires the use of square brackets [].

5. The square brackets [] are collectively an operator in Java, and are called the index operator. They are similar to the parentheses as they have the highest level of precedence compared to all other operators.
6. The index operator performs automatic bounds checking. Bounds checking makes sure that the index is within the range for the array being referenced. Whenever a reference to an array element is made, the index must be greater than or equal to zero and less than the size of the array. If the index is not valid, the exception `ArrayIndexOutOfBoundsException` is thrown.

B. Array Declarations and Memory Allocation

page 4 of 10

1. Array declarations look like this:

```
type[] arrayName;
```

This tells the compiler that `arrayName` will be used as the name of an array containing `type`. However, the actual array is not constructed by this declaration. Often an array is declared and constructed in one statement like this:

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

This tells the compiler that `arrayName` will be used as the name of an array containing `type`, and constructs an array object containing `length` number of slots.

2. An array is an object, and like any other object in Java is constructed out of main storage as the program is running. The array constructor uses different syntax than most object constructors; `type[length]` names the type of data in each slot and the number of slots. For example:

```
int[] list = new int[6];  
double[] data = new double[1000];  
Student[] school = new Student[1250];
```

Once an array has been constructed, the number of slots it has does not change.

3. The size of an array can be defined by using a `final` value.

```
final int MAX = 200;
int[] numb = new int[MAX];
```

4. When an array is declared, enough memory is allocated to set up the full size of the array.

C. Application of Arrays

page 5 of 10

1. Suppose we have a text file *votes.txt* of integer data containing all the votes cast in an election. This election happened to have three candidates and the values in the integer file are 1, 2, or 3, each corresponding to one of the three candidates.

Code Sample 16-2:

```
FileInput inFile = new FileInput("votes.txt");

int vote, total = 0, loop;

// sized to 4 boxes, initialized to 0's
int[] data = new int[4];

vote = inFile.readInt();
while (inFile.hasMoreTokens()){
    data[vote]++;
    total++;
    vote = inFile.readInt();
}
System.out.println("Total # of votes = " + total);
for (loop = 1; loop <= 3; loop++){
    System.out.println("Votes for #" + loop +
        " = " + data[loop]);
}
```

- a. The array *data* consists of four cells, each holding an integer value. The first cell, *data[0]*, is allocated but not used in this problem. After processing the entire file, the variable *data[n]* contains the number of votes for candidate *n*. We could have stored the information for candidate 1 in position 0, candidate 2 in position 1, and so forth, but the code is easier to follow if we can use a direct correspondence.

data

0	75	32	19
data[0]	data[1]	data[2]	data[3]

b. The value of vote is used to increment the appropriate cell of the array by +1.

2. A second example counts the occurrence of each alphabet letter in a text file.

Code Sample 16-3:

```
FileInput inFile = new FileInput("sample.txt");

int[] letters = new int[26]; // use positions 0..25
// to count letters
int total = 0;
char ch;

while (inFile.hasMoreLines()){
    String line = inFile.readLine().toLowerCase();
    for(int index = 0; index < line.length(); index++){
        ch = line.charAt(index);
        // line.charAt is from chn.util. It
        //extracts the entry.

        if ('a' <= ch && ch <= 'z') { // if letter
            letters[ch - 'a']++;
            total++;
        }
    }
}
System.out.println("Count letters");
System.out.println();
ch = 'a';
for (int loop = 0; loop < 26; loop++){
    System.out.println(ch + " : " + letters[loop]);
    ch++;
}
System.out.println();
System.out.println("Total letters = " + total);
```

a. Each line in the text file is read in and then each character in the line is copied into ch. If ch is an uppercase letter, it is converted to its lowercase counterpart.

b. If the character is a letter, the ASCII value of the letter is adjusted to fit the range

from 0-25. For example, if `ch == 'b'`, the program calculates `'b' - 'a' = 1`. Then the appropriate cell of the array is incremented by one.

D. Arrays as Parameters

page 6 of 10

1. The program *ArrayOps.java*, provides examples of passing arrays as parameters. Notice that the `final` integer constant `MAX = 6` is used to size the array in this program.

See [Handout 16.1, Example Program](#) - Arrays as Parameters *ArrayOps.java*

2. The `main` method declares an array named `data`. The array is initialized with the values 0..5 inside the `main` method.
3. The parameters of the `squareList` and `printList` methods are references to an array object. Any local reference to array `list` inside the `squareList` or `printList` methods is an alias for the array `data` inside of the `main` method. Notice that after the call of `squareList`, the values stored in array `data` in the `main` method have been permanently changed.
4. When the `rotateList` method is called, the `copy` method of the `ArrayOps` class is invoked and the local array `listCopy` is created as a copy of the array `data` in the `main` method.
5. The `rotateList` method rotates the values one cell to the right, with the last value moved to the front of the list. A call to `printList` is made inside the `rotateList` method just before leaving the method. After returning to the `main` method, notice that the array `data` is unchanged.

E. Arrays and Algorithms

page 7 of 10

In the following list, we introduce five important algorithms that are quite common in programs that analyze data in arrays. You will meet these again in later

lessons and labs.

1. Insertion is a standard problem that must be solved for all data structures. Suppose an array had 10 values and an 11th value was to be added. We are assuming the array can store at least 11 values.
 - a. If we could place the new value at the end, there would be no problem.
 - b. But if the new value must be inserted at the beginning of the list in position 0, the other 10 values must be moved one cell over in the list.
2. Deletion of a value creates an empty cell that probably must be dealt with. The most likely solution, after deleting a value, is to move all values that are to the right of the empty spot one cell to the left.
3. A traversal of an array consists of visiting every cell location, probably in order. The visit could involve printing out the array, initializing the array, finding the largest or smallest value in the array, etc.
4. Sorting an array means to organize values in either ascending or descending order. These algorithms will be covered in depth in future lessons.
5. Searching an array means to find a specific value in the array. There are several standard algorithms for searching an array. These will be covered in future lessons.

Summary/Review

page 8 of 10

Arrays are useful data structures and you will have many opportunities to program with them.

Chapter 17

A17 Introduction

page 1 of 11

In this lesson, you will learn about three sorting algorithms: bubble, selection, and insertion. You are responsible for knowing how they work, but you do not necessarily need to memorize and reproduce the code. After counting the number of steps of each algorithm, you will have a sense of the relative speeds of these three sorts.

The key topics for this lesson are:

- A. [Sorting Template Program](#)
- B. [Bubble Sort](#)
- C. [Selection Sort](#)
- D. [Insertion Sort](#)
- E. [Counting Steps - Quadratic Algorithms](#)
- F. [Animated Sort Simulations](#)
- G. [Sorting Objects](#)

[A17 Vocabulary](#)

page 2 of 11

BUBBLE SORT

NONDECREASING ORDER

SELECTION SORT

SWAP

INSERTION SORT

QUADRATIC

STUB

A. Sorting Template Program

page 3 of 11

1. A program shell has been provided in the curriculum as SortStep.java (the main test method), and SortsTemplate.java (the sort class template).

See [SortStep.java](#) and [SortsTemplate.java](#)

2. The program asks the user to select a sorting algorithm, fills the array with an amount of data chosen by the user, calls the sorting algorithm, and prints out the data after it has been sorted.

3. At this point, each sorting algorithm has been left as a method stub. A stub is an incomplete routine that can be called but does not do anything yet. The stub will be filled in later as each algorithm is developed and understood.
4. Stub programming is a programming strategy. It allows for the coding and testing of algorithms in the context of a working program. As each sorting algorithm is completed, it can be added to the program shell and tested without having to complete the other sections.
5. This stepwise development of programs using stub programming will be used extensively in future lessons.

B. Bubble Sort

page 4 of 11

1. Bubble Sort is the simplest of the three sorting algorithms, and also the slowest. The Bubble Sort gets its name from the way that the largest items “bubble” to the top (end). The procedure goes like this.
 - a. Move the largest remaining item in the current pass to the end of the data as follows. Starting with the first two items, swap them if necessary so that the larger item is after the smaller item. Now move over one position in the list and compare to the next item. Again swap the items if necessary.
 - b. Remove the largest item most recently found from the data to be searched and perform another pass with this new data at step a.
 - c. Repeat steps a and b above until the number of items to be searched is one.

To see how Bubble Sort works, let's try an example:

Steps	Data for pass	Sorted data
<u>Start pass 1</u> : compare 4 & 1.	4 1 3 2	
4 > 1 so swapped, now compare 4 & 3.	1 4 3 2	
4 > 3 so swapped, now compare 4 & 2.	1 3 4 2	
4 > 2 so swapped, end of pass.	1 3 2 4	
<u>Start pass 2</u> : compare 1 & 3.	1 3 2	4

3 > 1 so no swap, now compare 3 & 2.	1 3 2	4
3 > 2 so swapped, end of pass.	1 2 3	4
<u>Start pass 3</u> : now compare 1 & 2.	1 2	3 4
2 > 1 so no swap.	1 2	3 4
Only one item in this pass so it is done.	1	2 3 4
Done.		1 2 3 4

2. The following program implements the Bubble Sort algorithm.

```

void bubbleSort(ArrayList <Integer> list){
    for (int outer = 0; outer < list.size() - 1; outer++){
        for (int inner = 0; inner < list.size()-outer-1; inner++){
            if (list.get(inner) > list.get(inner + 1)){
                //swap list[inner] & list[inner+1]
                int temp = list.get(inner);
                list.set(inner, list.get(inner + 1));
                list.set(inner + 1, temp);
            }
        }
    }
}

```

3. Given a list of 6 values, the loop variables outer and inner will evaluate as follows.

When outer =	inner ranges from 0 to < (6 - outer - 1)
0	0 to 4
1	0 to 3
2	0 to 2
3	0 to 1
4	0 to 0

4. When outer = 0, then the inner loop will do 5 comparisons of pairs of values. As inner ranges from 0 to 4, it does the following comparisons:

inner	if (list.get(inner) > list.get(inner + 1))
0	if list[0] > list[1]
1	if list[1] > list[2]

...	...
4	if list[4] > list[5]

5. If `(list.get(inner) > list.get(inner + 1))` is **true**, then the values are out of order and a swap takes place. The swap takes three lines of code and uses a temporary variable `temp`.
6. After the first pass (`outer = 0`), the largest value will be in its final resting place (and may it rest in peace). When `outer = 1`, the `inner` loop only goes from 0 to 3 because a comparison between positions 4 and 5 is unnecessary. The `inner` loop is shrinking.
7. Because of the presence of duplicate values, this algorithm will result in a list sorted in non-decreasing order.
8. Here is a small list of data to test your understanding of Bubble Sort. Write in the correct sequence of integers after each advance of `outer`. (Answers are found in [Lesson A17 Handout](#), *Sorting Answers*.)

outer	57	95	88	14	25	6
1	_____	_____	_____	_____	_____	_____
2	_____	_____	_____	_____	_____	_____
3	_____	_____	_____	_____	_____	_____
4	_____	_____	_____	_____	_____	_____
5	_____	_____	_____	_____	_____	_____

C. Selection Sort

page 5 of 11

1. The Selection Sort also makes several passes through the list. On each pass, it compares each remaining item to the smallest (or largest) item that has been found so far in the pass. In the example below, the Selection Sort method finds the smallest item on each pass. At the end of a pass, the smallest item found is swapped with the last remaining item for that pass. Thus, swapping only occurs once for each pass. Reducing the number of swaps makes the algorithm more efficient.

2. The logic of Selection Sort is similar to Bubble Sort except that fewer swaps are executed.

```
void selectionSort(ArrayList <Integer> list){
    int min, temp;

    for (int outer = 0; outer < list.size() - 1; outer++){
        min = outer;
        for (int inner = outer + 1; inner < list.size(); inner++){
            if (list.get(inner) < list.get(min)) {
                min = inner; // a new smallest item is found
            }
        }
        //swap list[outer] & list[min]
        temp = list.get(outer);
        list.set(outer, list.get(min));
        list.set(min, temp);
    }
}
```

3. Again, assuming that we have a list of 6 numbers, the outer loop will range from 1 to 5. When `outer = 1`, we will look for the smallest value in the list and move it to the first position in the array.
4. However, when looking for the smallest value to place in position 1, we will not swap as we search through the list. The algorithm will check from indexes 1 to 5, keeping track of where the smallest value is found by saving the index of the smallest value in `min`. After we have found the location of the smallest value, we swap `list[outer]` and `list[min]`.
5. By keeping track of where the smallest value is located and swapping only once, we have a more efficient algorithm than Bubble Sort.
6. Here is a small list of numbers to test your understanding of Selection Sort. Fill in the correct numbers for each line after the execution of the outer loop. (Answers are found in [Lesson A17 Handout](#), *Sorting Answers*.)

outer	57	95	88	14	25	6
1	_____	_____	_____	_____	_____	_____
2	_____	_____	_____	_____	_____	_____
3	_____	_____	_____	_____	_____	_____
4	_____	_____	_____	_____	_____	_____
5	_____	_____	_____	_____	_____	_____

1. Insertion Sort takes advantage of the following fact.

If $A < B$ and $B < C$, then it follows that $A < C$. We can skip the comparison of A and C .

2. Consider the following partially sorted list of numbers.

2 5 8 3 9 7

The first three values of the list are sorted. The 4th value in the list, (3), needs to move back in the list between the 2 and 5.



This involves two tasks, finding the correct insert point and a right shift of any values between the start and insertion point.

3. The code follows.

```
void insertionSort(ArrayList <Integer> list){
    for (int outer = 1; outer < list.size(); outer++){
        int position = outer;
        int key = list.get(position);

        // Shift larger values to the right
        while (position > 0 && list.get(position - 1) > key){
            list.set(position, list.get(position - 1));
            position--;
        }
        list.set(position, key);
    }
}
```

4. By default, a list of one number is already sorted. Hence the outer loop skips position 0 and ranges from positions 1 to `list.size()`. For the sake of discussion, let us assume a list of 6 numbers.
5. For each pass of outer, the algorithm will determine two things concerning the value stored in `list[outer]`. First, it finds the location where `list[outer]` needs to be inserted in

the list. Second, it does a right shift on sections of the array to make room for the inserted value if necessary.

6. Constructing the inner `while` loop is an appropriate place to apply DeMorgan's laws:

1. The inner `while` loop postcondition has two possibilities:

The value (key) is larger than its left neighbor.

The value (key) moves all the way back to position 0.

2. This can be summarized as:

```
(0 == position || list.get(position - 1) <= key)
```

3. If we negate the loop postcondition, we get the while loop boundary condition:

```
(0 != position && list.get(position - 1) > key)
```

4. This can also be rewritten as:

```
((position > 0) && (list.get(position - 1) > key))
```

7. The two halves of the boundary condition cover these situations:

`(position > 0)` -> we are still within the list, keep processing

`list[position - 1] > key` -> the value in `list[pos-1]` is larger than key, keep moving left (`position--`) to find the first value smaller than key.

8. The Insertion Sort algorithm is appropriate when a list of data is kept in sorted order with infrequent changes. If a new piece of data is added, probably at the end of the list, it will get quickly inserted into the correct position in the list. Many of the other values in the list do not move and the inner `while` loop will not be used except when inserting a new value into the list.

9. Here is the same list of six integers to practice Insertion Sort. (Answers are found in [Lesson A17 Handout](#), *Sorting Answers*.)

outer	57	95	88	14	25	6
1	_____	_____	_____	_____	_____	_____
2	_____	_____	_____	_____	_____	_____
3	_____	_____	_____	_____	_____	_____
4	_____	_____	_____	_____	_____	_____
5	_____	_____	_____	_____	_____	_____

1. These three sorting algorithms are categorized as quadratic sorts because the number of steps increases as a quadratic function of the size of the list.
2. It will be very helpful to study algorithms based on the number of steps they require to solve a problem. We will add code to the sorting template program and count the number of steps for each algorithm.
3. This will require the use of an instance variable - we'll call it `steps`. The `steps` variable will be maintained within the sorting class and be accessed through appropriate accessor and modifier methods. You will need to initialize `steps` to 0 at the appropriate spot in the main menu method.
4. For our purposes, we will only count comparisons of items in the list, and gets or sets within the list. These operations are typically the most expensive (time-wise) operations in a sort.
5. As you type in the sorting algorithms, add increment statements for the instance variable `steps`. For example, here is a revised version of the `bubbleSort` method:

```
public void bubbleSort(ArrayList <Comparable> list){
    steps = 0;
    for (int outer = 0; outer < list.size() - 1; outer++){
        for (int inner = 0; inner < list.size()-outer-1; inner++){
            steps += 3;//count one compare and 2 gets
            if (list.get(inner).compareTo(list.get(inner + 1)) > 0){
                steps += 4;//count 2 gets and 2 sets
                Comparable temp = list.get(inner);
                list.set(inner,list.get(inner + 1));
                list.set(inner + 1,temp);
            }
        }
    }
}
```

6. It is helpful to remember that a `for` statement is simply a compressed `while` statement. Each `for` loop has three parts: initialization, boundary check, and incrementation.
7. As you count the number of steps, an interesting result will show up in your data. As the size of the data set doubles, the number of steps executed increases by approximately

four times, a “quadratic” rate.

8. Bubble Sort is an example of a quadratic algorithm in which the number of steps required increases at a quadratic rate as the size of the data set increases.
9. A quadratic equation in algebra is one with a squared term, like x^2 . In our sorting example, as the size of the array increases to N , the number of steps required for any of the quadratic sorts increases as a function of N^2 .

F. Animated Sort Simulations

page 8 of 11

1. Rameen Mohammadi of the Computer Science Department at SUNY Oswego has created a web applet located at <http://www.cs.oswego.edu/~mohammad/classes/csc241/samples/sort/Sort2-E.html> that provides a nice illustration of animated sorting simulations.

G. Sorting Objects

page 9 of 11

1. Notice that the sorts we developed above know how to compare Integers. Comparison is built into the Integer class. What if we wanted to write a sort that could work on Strings? You cannot use ‘<’ on Strings. Remember you have to use the compareTo method.
2. To convert the BubbleSort, make the following changes that are highlighted in yellow.

```
void bubbleSort(ArrayList <String> list){
    for (int outer = 0; outer < list.length - 1; outer++){
        for (int inner = 0; inner < list.size()-outer-1; inner++){
            if (list.get(inner).compareTo(list.get(inner + 1)) > 0){
                //swap list[inner] & list[inner+1]
                String temp = list.get(inner);
                list.set(inner, list.get(inner + 1));
                list.set(inner + 1, temp);
            }
        }
    }
}
```

3. If I am able to sort my data, there must be an order defined for it. Classes that have an order should have a `compareTo` method. Java defines an Interface, `Comparable`, just for this purpose (see below for some information on `Comparable`). To make a `BubbleSort` that will work on any objects that implement `Comparable`, make the following changes, again highlighted in yellow.

```
void bubbleSort(ArrayList <Comparable> list){
    for (int outer = 0; outer < list.length - 1; outer++){
        for (int inner = 0; inner < list.size()-outer-1; inner++){
            if (list.get(inner).compareTo(list.get(inner + 1)) > 0){
                //swap list[inner] & list[inner+1]
                Comparable temp = list.get(inner);
                list.set(inner, list.get(inner + 1));
                list.set(inner + 1, temp);
            }
        }
    }
}
```

Now this method is quite reusable because we can use it to sort any `Comparable` object. The `compareTo` interface follows.

```
Interface java.lang.Comparable
int compareTo(Object other)
    // returns value < 0 if this is less than other
    // returns value = 0 if this is equal to other
    // returns value > 0 if this is greater than other
```

Remember to consider whether or not it makes sense to compare objects that you build. If it does, implement the `Comparable` Interface. It would also make sense to provide an `equals` method for your class.

Sorting data is one of the best applications of computers and software. What takes hours or days by hand can be sorted in seconds or minutes by a computer. However, these quadratic algorithms have problems sorting large amounts of data. More efficient sorting algorithms will be covered in later lessons.

Chapter 18

In Lesson A17, *Quadratic Sorting Algorithms*, we saw how the number of steps required increased N^2 when sorting N elements. In this lesson, we will study a recursive sort, called mergeSort that works by dividing lists in half. After solving a preliminary merge problem, you will code a recursive mergeSort.

The key topics for this lesson are:

- A. [Non-Recursive MergeSort](#)
- B. [A Merge Algorithm](#)
- C. [Recursive MergeSort](#)
- D. [Order of Recursive MergeSort](#)

[A18 Vocabulary](#)

MERGE

MERGESORT

A. A Non-Recursive MergeSort

1. The overall logic of mergeSort is to "divide and conquer." A list of random integers will be split into two or more equal-sized lists (each with the same number of elements, plus or minus one), with each partial list or "sublist" sorted independently of the other. The next step will be to merge the two sorted sublists back into one big sorted list.

2. Here is a non-recursive mergeSort method. We divide the list into two equal-sized parts and sort each with the selection sort, then merge the two using an algorithm to be discussed in part B.

```
/* List A is unsorted, with A.size() values in the ArrayList.  
first is the index of the first value; last  
is the index of the last value in the ArrayList;  
first < last.  
*/  
void mergeSort (ArrayList <Integer> A, int first, int last){  
    int mid;  
  
    mid = (first + last) / 2;  
    selectionSort (A, first, mid);  
    selectionSort (A, mid+1, last);  
    merge (A, first, mid, last);  
}
```

3. A modified selection sort will have to be written to sort a range of values in list A. Likewise, the merge method will have to be modified to internally merge two halves of the array into one ordered array.

See [Transparency A18.1](#), *MergeSort Example*

4. The following example will illustrate the action of a non-recursive mergeSort on a section of a list containing 8 values:

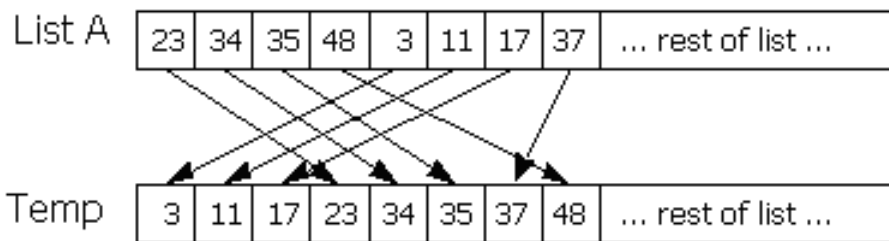
Unsorted List

34	23	48	35	37	3	11	17	... rest of list ...
----	----	----	----	----	---	----	----	----------------------

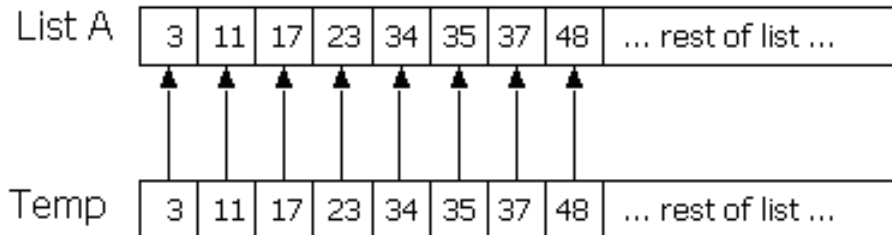
List After Sorting Each Half

23	34	35	48	3	11	17	37	... rest of list ...
----	----	----	----	---	----	----	----	----------------------

5. Merging the two halves of the array in the modified merge method will require the use of a local temporary array. Because the two sublists are stored within one array, the easiest approach is to merge the two sublists into another array, then copy the temp array back to the original.



Then copy Temp back into List A:



6. This version of merge will need to be able to work with sections of ArrayList A. Here is a proposed parameter list for merge:

```
/* will merge the two sorted sublists within A into
one continuous sublist from A[first] .. A[last].
The left list ranges from A[first]..A[mid]. The
right list ranges from A[mid+1]..A[last].
*/
void merge (ArrayList <Integer> A, int first, int mid, int last)
```

7. The recursive version of mergeSort will require the above version of merge. However, to help you understand how to write a merge method, we next present a simpler merge algorithm.

B. A Merge Algorithm

page 4 of 9

1. The mergeSort algorithm requires a merge algorithm that we will design first.
2. The method header and the specifications of the merge method are given below. You may assume the ArrayList definitions from the sorting template program in Lesson 17 apply.

```
/* Preconditions: Lists A and B are non-empty and in sorted nondecreasing
order.
Action: Lists A and B are merged into one ArrayList, C.
Postcondition: List C contains all the values from
```

Lists A and B, in nondecreasing order.

*/

```
void merge (ArrayList <Integer> A, ArrayList <Integer> B, ArrayList  
<Integer> C)
```

3. The merge method breaks down into four cases:
 - a. ArrayList A is done, so pull a value from ArrayList B.
 - b. ArrayList B is done, so pull a value from ArrayList A.
 - c. Neither is done, and if $A[i] < B[j]$ (where i & j are index markers in lists A and B) then pull from ArrayList A.
 - d. Neither is done, and if $B[j] \leq A[i]$ then pull from ArrayList B.
4. It is important to deal with the four cases in the order described above. For example, if you are done with ArrayList A (if $i > A.length-1$), you do not want to inspect any values past $A[i]$.
5. Example of method merge results:

See [Transparency A18.2](#), *Merging Two Lists*

A: 2 6 11 15 21

B: 4 5 9 13 17 25 29

C: 2 4 5 6 9 11 13 15 17 21 25 29

C. Recursive MergeSort

page 5 of 9

1. Instead of dividing the list once, a recursive mergeSort will keep dividing the list in half until the sublists are one or two values in length.
2. When developing a recursive solution, a key step is identifying the base case of the solution. What situation will terminate the recursion? In this case, a sublist of one or two values will be our two base cases.
3. Let's try and work through the recursive mergeSort of a list of eight values.

16	91	77	45	5	88	65	21
----	----	----	----	---	----	----	----

The list is divided into two sublists:

16	91	77	45
----	----	----	----

5	88	65	21
---	----	----	----

Now let's work on the left sublist. It will be divided into lists of two.

16	91
----	----

77	45
----	----

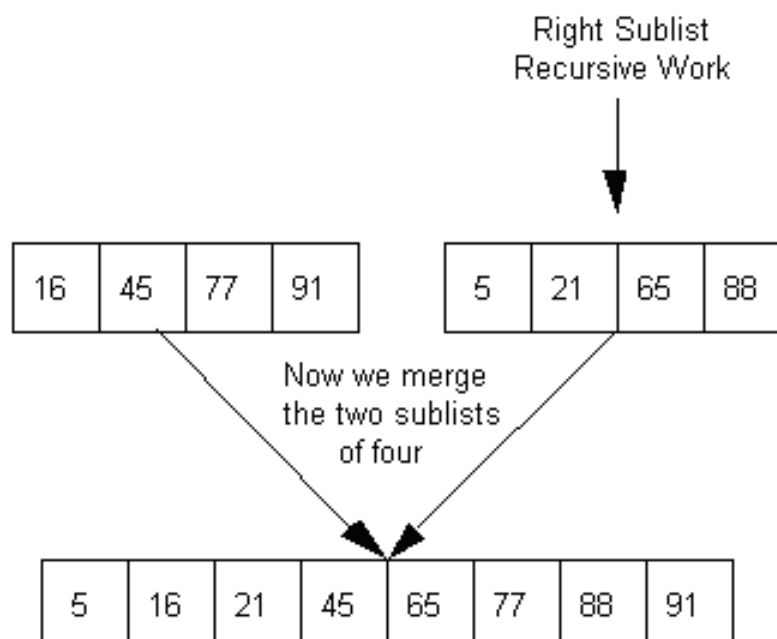
Each list of two is now very easy to sort. After each list of two is sorted, we then merge these two lists back into a list of four.

16	91
----	----

45	77
----	----

16	45	77	91
----	----	----	----

Now the algorithm proceeds to solve the right sublist (positions 5-8) recursively. Then the two lists of four are merged back together.



1. Suppose that we have a list of 8 numbers. If we trace the migration of one value, it will be a member of the following sizes of lists: eight, four, two. The number of calls of `mergeSort` needed to sort one value into its final resting spot is $\log_2 N$. If $N = 8$, then it will take three calls of the algorithm for one value to find its final resting spot.
2. We must apply $\log_2 N$ steps to sort N elements in the list. The order of recursive `Mergesort` is $O(N * \log_2 N)$ or $O(N * \log N)$.
3. What about the cost of merging the fragments of the list? The merge algorithm is a linear one, so when combined with the `mergeSort` routine, we have a $O(N * \log N) + O(N)$, which remains in the category of an $O(N * \log N)$ algorithm.

The recursive `mergeSort` produces a dramatic increase in efficiency in comparison with the N^2 order of the quadratic sorts. This concept of dividing the problem in two is used in several other classic algorithms. Once again, recursion makes it easier to define a problem and code the solution.

Chapter 19

Searching for an item is a very important algorithm to a computer scientist. What makes

computers tremendously valuable is their ability to store and search for information quickly and efficiently. For example, Internet search engines process billions of pages of information to help determine the most appropriate resources for users, and a word processor's spell-checking feature enables quick searching of large dictionaries. In this lesson, you will learn about a simple sequential search and the very efficient binary search.

The key topics for this lesson are:

- A. [Sequential Search](#)
- B. [Binary Search](#)
- C. [Recursive vs. Non-recursive Algorithms](#)

[A19 Vocabulary](#)

page 2 of 9

BINARY SEARCH

SEQUENTIAL SEARCH

A. Sequential Search

page 3 of 9

1. Searching a linear data structure, such as an array, can be as simple and straightforward as checking through every value until you find what you are looking for. A sequential search, also known as a *linear search*, involves starting at the beginning of a list or sequence, sorted or not, and searching one-by-one through each item, in the order they exist in the list, until the value is found.
2. This unsophisticated approach is appropriate for small lists or unordered lists.
3. The order of a sequential search is linear, $O(N)$.

1. The word binary (from the word two) refers to anything with two possible options or parts. A binary search involves binary decisions - decisions with two choices.
2. The underlying idea of a binary search is to divide one's data in half and to examine the data at the point of the split. If the data is sorted, it's very easy and efficient to ignore one half or the other half of the data, depending on where the value that is being searched is located.
3. Assuming that a list is already sorted, a target value is searched for by repeating the following steps:
 - a. Divide the list in half.
 - b. Examine the value in the middle of the list. Is the target value equal to the value there, or does it come before or after the center value? If the target value comes before or after, then return to step a, to repeat the process with the halved list where the target value is located.
4. For example, with a list of 1,024 sorted values, we happen to be searching for a target value that is stored in position 492. Using a binary search, the list of 1,024 is split in half; because the target value is not found in position 512, we proceed to search the first half (and discard the values in the second half). Within the sublist from 1...512, we do another binary search sequence: split; examine; and binary search again.
5. The speed of a binary search comes from the elimination of half of the data set each time. If each arrow below represents one binary search process, only ten steps are required to search a list of 1,024 numbers:

1024 → 512 → 256 → 128 → 64 → 32 → 16 → 8 → 4 → 2 → 1

The $\log_2 1024$ is 10. In a worst-case scenario, if the size of the list doubled to 2,048, only one more step would be required using a binary search.

The efficiency of a binary search is illustrated in this comparison of the number of entries in a list and the number of binary divisions required.

Number of Entries	Number of Binary Divisions
1,024	10

2,048	11
4,096	12
...	...
32,768	15
...	...
1,048,576	20
N	$\log_2 N$

6. The order of a binary search is $O(\log_2 N)$.

C. Recursive vs. Non-recursive Algorithms

page 5 of 9

1. The binary search algorithm can be coded recursively or non-recursively. Here are some arguments for each method.
2. A non-recursive version requires less memory and fewer steps by avoiding the overhead of making recursive calls.
3. However, the recursive version is somewhat easier to understand and code and is more fun! The lab assignment can be coded as either a recursive or non-recursive version of binary search.

Summary/Review

page 6 of 9

Searching algorithms is widely used in programs. Binary searching is the fastest - if the list is sorted.

Chapter 20

A class represents a set of objects that share the same structure and behaviors. The class determines the structure of objects by specifying variables that are contained in each instance of the class, and it determines behavior by providing the instance methods that express the behavior of the objects. This is a powerful idea. However, something like this can be done in most programming languages. The central new idea in object-oriented programming is to allow classes to express the similarities among objects that share some, but not all, of their structure and behavior. Such similarities can be expressed using inheritance and polymorphism.

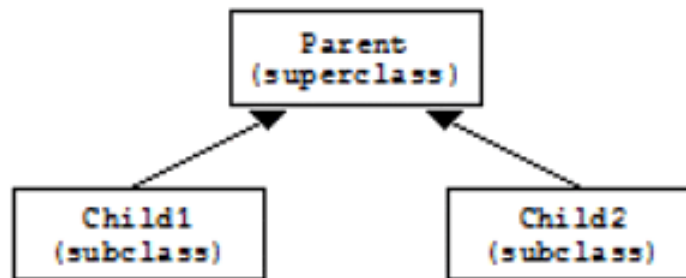
The key topics for this lesson are:

- A. [Inheritance](#)
- B. [Abstract Classes](#)
- C. [Polymorphism](#)
- D. [Interfaces](#)

[A20 Vocabulary](#)

ABSTRACT	CONCRETE CLASS
EARLY BINDING	INSTANCE
INTERFACE	LATE BINDING
POLYMORPHISM	SUBCLASS
SUPERCLASS	

1. A key element in Object Oriented Programming (OOP) is the ability to derive new classes from existing classes by adding new methods and redefining existing methods. The new class can inherit many of its attributes and behaviors from the existing class. This process of deriving new classes from existing classes is called inheritance, which was introduced in Lesson A11, Inheritance.



The more general class that forms the basis for inheritance is called the superclass. The more specialized class that inherits from the superclass is called the subclass (or derived class).

2. In Java, all classes belong to one big hierarchy derived from the most basic class, called Object. This class provides a few features common to all objects; more importantly, it makes sure that any object is an instance of the Object class, which is useful for implementing structures that can deal with any type of object. If we start a class from “scratch,” the class automatically extends Object. For example:

```
public class SeaCreature{  
    ...  
}
```

is equivalent to:

```
public class SeaCreature extends Object{  
    ...  
}
```

when new classes are derived from seacreature, a class hierarchy is created. For example:

```
public class Fish extends SeaCreature{  
    ...  
}  
  
public class Mermaid extends SeaCreature{  
    ...  
}
```

This results in the hierarchy shown below.

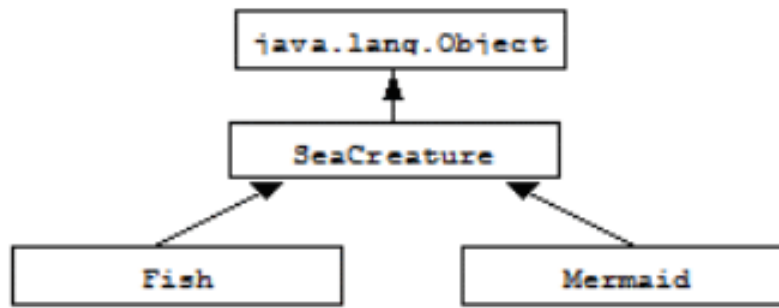


Figure 20-1. SeaCreature and two derived classes

B. Abstract Classes

page 4 of 8

1. The classes that lie closer to the top of the hierarchy are more general and abstract; the classes closer to the bottom are more specialized. Java allows us to formally define an *abstract* class. In an abstract class, some or all methods are declared **abstract** and left without code.
2. An **abstract** method has only a heading: a declaration that gives the method's name, return type, and arguments. An **abstract** method has no code. For example, all of the methods in the definition of the SeaCreature class shown below are abstract. SeaCreature tells us what methods a sea creature must have, but not how they work.

```
// A type of creature in the sea
public abstract class SeaCreature{
    // Called to move the creature in its environment
    public abstract void swim();

    // Attempts to breed into neighboring locations
    public abstract void breed();

    // Removes this creature from the environment
    public abstract void die();

    // Returns the name of the creature
    public abstract String getName();
}
```

3. In an **abstract** class, some methods and constructors may be fully defined and have code supplied for them while other methods are **abstract**. A class may be declared abstract for other reasons as well. For example, some of the instance variables in an abstract class may belong to abstract classes.
4. More specialized subclasses of an abstract class have more and more methods

defined. Eventually, down the inheritance line, the code is supplied for all methods. A class where all the methods are fully defined is called a *concrete* class. A program can only create objects of concrete classes. An object is called an *instance* of its class. An **abstract** class cannot be instantiated.

5. Different concrete classes in the same hierarchy may define the same method in different ways. For example:

```
public class Fish extends SeaCreature{
    ...

    /**
     * Returns the name of the creature
     */
    public String getName(){
        return "Wanda the Fish";
    }
    ...
}

public class Mermaid extends SeaCreature{
    ...

    /**
     * Returns the name of the creature
     */
    public String getName(){
        return "Ariel the Mermaid";
    }
    ...
}
```

C. Polymorphism

page 5 of 8

1. In addition to facilitating the re-use of code, inheritance provides a common base data type that lets us refer to objects of specific types through more generic types of references; in particular, we can mix objects of different subtypes in the same collection. For example:

```
SeaCreature s = new Fish(...);
...
s.swim();
```

The data type of an instance of the `Fish` class is a `Fish`, but it is also a kind of

SeaCreature. Java provides the ability to refer to a specific type through more generic types of references.

2. There may be situations that require a reference to an object using its more generic supertype rather than its most specific type. One such situation is when different subtypes of objects in the same collection (array, list, etc.) are mixed. For example:

```
ArrayList <SeaCreature> creature = new ArrayList <SeaCreature>;
Creature.add( new Fish(...));
Creature.add( new Mermaid(...));
...
creature.get(currentCreature).swim();
```

This is possible because both `Fish` and `Mermaid` are `SeaCreatures`.

3. Note that the `Fish` and `Mermaid` classes provide two different implementations of the `swim` method. The correct method that belongs to the class of the actual object is located by the Java virtual machine. That means that one method call

```
String s = x.getname();
```

can call different methods depending on the current reference of `x`. When a subclass redefines the implementation of a method from its superclass, it is called overriding the method. Note that for overridden methods in Java, the actual method to call is always determined at run time. This is called *dynamic binding* or *late binding*.

4. The principle that the actual type of the object determines the method to be called is *polymorphism* (Greek for “many shapes”). The same computation works for objects of many forms and adapts itself to the nature of the objects.
5. Overloading a method is often confused with overriding a method because the names are so similar. Overloading a method means to keep the name of the method, but change the parameter list. In this case, the compiler can choose the method that is actually called because the signatures are different. The `Math` class has many examples of overloaded methods. For example `Math.max(double a, double b)` and `Math.max(int a, int b)` are overloaded versions of the `max` method. The compiler determines which one to call depending on the type of the arguments that are being passed in.

1. In Lesson A14, *Boolean Algebra*, it was learned that Java has a class-like form called an *interface* that can be used to encapsulate only abstract methods and constants. An interface can be thought of as a blueprint or design specification. A class that uses this interface is a class that *implements the interface*.
2. An interface is similar to an abstract class: it lists a few methods, giving their names, return types, and argument lists, but does not give any code. The difference is that an abstract class may have its constructors and some of its methods implemented, while an interface does not give any code for its methods, leaving their implementation to a class that implements the interface.
3. `interface` and `implements` are Java reserved words. Here is an example of a simple Java interface:

```
public interface Comparable{  
    public int compareTo(Object other);  
}
```

This looks much like a class definition, except that the implementation of the `compareTo()` method is omitted. A class that implements the `Comparable` interface must provide an implementation for this method. The class can also include other methods and variables. For example,

```
class Location implements Comparable{  
    public int compareTo(Object other){  
        . . . // do something -- presumably, compare objects  
    }  
    . . . // other methods and variables  
}
```

Any class that implements the `Comparable` interface defines a `compareTo()` instance method. Any object created from such a class includes a `compareTo()` method. We say that an object implements an interface if it belongs to a class that implements the interface. For example, any object of type `Location` implements the `Comparable` interface. Note that it is not enough for the object to include a `compareTo()` method. The class that it belongs to must say that it “implements” `Comparable`.

4. A class can implement any number of interfaces. In fact, a class can both extend another class and implement one or more interfaces. So, we can have things like:

```
class Fish extends SeaCreature implements Locatable, Eatable{  
    . . .  
}
```

5. An interface is very much like an abstract class. It is a class that can never be used for constructing objects, but can be used as a basis for making subclasses. Even though you can't construct an object from an interface, you can declare a variable whose type

is given by the interface. For example, if `Locatable` is an interface defined as follows

```
public interface Locatable{
    Location location();
}
```

then if `Fish` and `Mermaid` are classes that implement `Locatable`, you could say

```
/**
 * Declare a variable of type Locatable. It can refer to
 * any object that implements the Locatable interface.
 */
Locatable nemo;

nemo = new Fish();    // nemo now refers to an object
                      // of type Fish
nemo.location();      // Calls location () method from
                      // class Fish
nemo = new Mermaid(); // Now, nemo refers to an object
                      // of type Mermaid
nemo.location();      // Calls location() method from
                      // class Mermaid
```

A variable of type `Locatable` can refer to any object of any class that implements the `Locatable` interface. A statement like `nemo.location()`, above, is legal because `nemo` is of type `Locatable`, and any `Locatable` object has a `location()` method.

6. You are not likely to need to write your own interfaces until you get to the point of writing fairly complex programs. However, there are a few interfaces that are used in important ways in Java's standard packages. You'll learn about some of these standard interfaces in the next few lessons, and you will see examples of interfaces in the Marine Biology Simulation, which was developed for use in AP Computer Science courses by the College Board™.

Summary/Review

page 7 of 8

The main goals of OOP are team development, software reusability, and easier program maintenance. The main OOP concepts that serve these goals are abstraction, encapsulation, inheritance, and polymorphism. In this lesson, we reviewed these key concepts and their implementation in Java. This lesson examined how Java uses classes and interfaces, inheritance hierarchies, and polymorphism to achieve the goal of better-engineered programs.

Chapter 21

Knowing how the computer stores information will give you an understanding of some of the common computational errors such as over-flow, round-off and, under-flow errors. This lesson is a very light introduction to number systems and their relationship to Computer Science.

The key topics for this lesson are:

- A. [Introduction to Number Systems](#)
- B. [Review Decimal](#)
- C. [Octal](#)
- D. [Hexadecimal](#)
- E. [Binary](#)
- F. [Short Cuts Between Binary, Octal, Hexadecimal](#)
- G. [How this Relates to Computers and Overflow Errors](#)

[A21 Vocabulary](#)

BINARY	DECIMAL
HEXADECIMAL	OCTAL
OVER-FLOW ERROR	ROUND-OFF ERROR
UNDER-FLOW ERROR	

1. Our number system has 10 as its base. This numbering system (decimal) may have arisen quite naturally from counting on our ten fingers. There are other common number bases that we use without thinking about them, such as when we deal with time (base 60 or sexagesimal): there are, 60 minutes in an hour, and 60 seconds in a minute. Aviators also use a different base when they are dealing with directions and degrees.
2. In Computer Science, we use binary (base 2), octal (base 8), and hexadecimal (base 16) quite a bit. After learning how the bases work, we will look into their relationship to modern computers and how computer calculations are performed.

B. Review Decimal

page 4 of 11

1. In reviewing our number system, let's look at what the number 1234 means to us. We are taught very early on about place values. In the decimal number system, starting at the rightmost digit we have the ones place, then the tens place, hundreds place, and finally the thousands place. We are only allowed the digits 0 through 9. The number 1234 represents the following components:

$$1*1000 + 2*100 + 3*10 + 4*1$$

or

$$1*10^3 + 2*10^2 + 3*10^1 + 4*10^0$$

We are very accustomed to using decimal numbers so we do not think about breaking a number into its components. However, it is important to learn and understand what numbers represent, often by using hands-on activities.

Example: Given an integer variable num, reverse the number. If we knew there were three digits, we could

```
int a = num/100;
int remainder = num%100;
int b = remainder/10;
int c = remainder%10;
//now we could reassemble it
num = c*100 + b*10 + a;
```

This is similar to what we did in Lab Assignment A3.2, *Coins*.

If we do not know how many digits are in the number, we could use a loop.

```
int x = 12345;
int newX = 0;
int temp = 0;
while(x > 0){
    temp = x%10;
    newX = newX * 10 + temp;
    x /= 10;
}
x = temp;
```

2. If we move to the right of the decimal place, we start using negative exponents.

10^3	10^2	10^1	10^0	10^{-1}	10^{-2}
--------	--------	--------	--------	-----------	-----------

C. Octal

page 5 of 11

1. After understanding how the decimal number system works, it is quite easy to learn other bases, such as base 8 (octal). They all work basically the same way. In base 8, the available digits are 0 through 7, and each number occupies a place value. The place values are:

8^3	8^2	8^1	8^0	8^{-1}	8^{-2}
-------	-------	-------	-------	----------	----------

We can designate that a number is in a base other than base 10 (decimal) by using subscripts. Note: If there is no subscript, the number is assumed to be decimal. Let's use the number 123.6 in base 8 - it must be written as 123.6_8 .

The number 123.6_8 can be converted to a decimal as:

$$1 \cdot 8^2 + 2 \cdot 8^1 + 3 \cdot 8^0 + 6 \cdot 8^{-1}$$

$$1 \cdot 64 + 2 \cdot 8 + 3 \cdot 1 + 6 \cdot (1/8)$$

$$83.75$$

Now, using the number 567, we'll show how to convert it from the decimal system to the octal system. Start by looking for the largest power of 8 less than 567. This would be 512 or 8^3 . So in the 8^3 place value, we put a 1. That leaves us with a remainder of 55. The next place value is 8^2 , but 64 is greater than 55 so that place holds a zero. The next place value is 8^1 . 55 divided by 8 gives 6 for the 8^1 place value, with a remainder of 7 left over for the next column. The leftover gets placed in the ones column.

8^3	8^2	8^1	8^0	8^{-1}	8^{-2}
1	0	6	7	0	0

or 1067_8

D. Hexadecimal

page 6 of 11

- Hexadecimal is base 16. This base works exactly the same as base 10 and base 8. Hexadecimal will allow the digits 0 through 15. This gets confusing, however, because the digits 10 through 15 are double digits and raises the question: is 10 a single value or two symbols? To avoid confusion we use other symbols, A through F, to represent the values of 10 through 15. Let's use the hexadecimal number 34CD and convert it to decimal. This number has the value of:

$$\begin{aligned}
 &3 \cdot 16^3 + 4 \cdot 16^2 + 12 \cdot 16^1 + 13 \cdot 16^0 \\
 &12288 + 1024 + 192 + 13 \\
 &13517
 \end{aligned}$$

So the hexadecimal number 34CD is equivalent to 13517 in base 10 (decimal).

Converting from decimal to hexadecimal is the same process used for octal earlier:

$$\begin{aligned}
 12547 &\rightarrow \\
 12547 / (16^3) &\rightarrow 3, \text{ remainder } 259 \\
 259 / (16^2) &\rightarrow 1, \text{ remainder } 3 \\
 3 / (16^1) &\rightarrow 0, \text{ remainder } 3
 \end{aligned}$$

So 12547 in the decimal number system is equivalent to 3103_{16} in the hexadecimal system.

1. Not surprisingly, binary works the same way as octal and hexadecimal. Binary is base 2, so the only digits that are used are 0 and 1, and the place values are all powers of 2. A binary digit is called a bit. The place values for binary are shown in the table below.

2^3	2^2	2^1	2^0	2^{-1}	2^{-2}
8	4	2	1	1/2	1/4

Let's convert two decimal numbers into the binary system: 13 and 482.

The decimal number 13 \rightarrow 1101_2 ($1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0$)

The decimal number 482 \rightarrow 111100010_2

Calculations for converting 482:

Find the largest power of two that divides into 482. So 2^8 , or 256.

$482/256 = 1$, subtract to get the leftover, 226.

Now check to see if 2^7 , 128 goes into 226. Notice we only have the choice of 1 or 0 so this will be simple to calculate. $226 - 128 \rightarrow 98$ (another 1)

$98 - 64 \rightarrow 34$ (another 1)

$34 - 32 \rightarrow 2$ (another 1)

can't subtract 16 (so a 0 goes here)

can't subtract 8 (another 0)

can't subtract 4 (another 0)

$2 - 2 \rightarrow 0$ (another 1)

can't subtract 1 (another 0)

Answer: 111100010_2

1. There is a relationship between binary, octal, and hexadecimal numbers. Binary is base 2, octal is base 8(2³) and hexadecimal is base 16(2⁴). Let's convert the binary number 101101011010111 to a hexadecimal number.

Binary -> hexadecimal
101101011010111

Starting at the rightmost digit, split the number into groups of 4
101 1010 1101 0111

2. Each of these groups has 4 binary digits that can range from 0 -15. This is exactly the value of one hexadecimal digit, so match each group of four bits with one hexadecimal digit.

Binary Number Groups	Hexadecimal Equivalent
101	5
1010	A
1101	D
0111	7

So our binary number is equivalent to 5AD7₁₆. Going from hexadecimal reverses the procedure so each hexadecimal digit expands to 4 bits.

The same process occurs between octal and binary using only 3 bits.

Try the following conversions for practice:

10 110 101₂ -> ____ _ _ _₈
3F₁₆ -> ____ _ _ _₂
352₈ -> ____ _ _ _₂
482 -> ____₂
482 -> ____₈
482 -> ____₁₆
10001₂ -> ____₁₀
5776₈ -> ____₁₀
3DB₁₆ -> ____₁₀
110 111 010₂ -> ____ _ _ _₈
1011 0010 1111₂ -> ____ _ _ _₁₆
3FA₁₆ -> ____ _ _ _₂
712₈ -> ____ _ _ _₂

The answers appear at the end of this lesson.

G. How this Relates to Computers and Overflow Errors

page 9 of 11

1. Modern computers are binary computers. They are made up of switches. Each switch is either on or off at any one time. Everything we put into the computer (numbers and letters) must be converted to binary numbers.
2. At some point in time when using a computer, there are not going to be enough bits to accurately represent the decimal numbers in the real world. We will look at this in a smaller scenario, in order to simplify this discussion.
3. Let's hypothesize that we have 8 bits for our numbers and there are no negative numbers.

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
128	64	32	16	8	4	2	1

4. The numbers available to us range from 0000 0000 to 1111 1111 (binary), which is 0 to 255 (decimal). This is not very large for an integer. Attempting to put a larger number into one of our integers would result in an overflow error. Look back at the charts in Lesson A3 for minimum and maximum values for the different types of primitive data types.
5. Another computer problem arises when using numbers that have decimal places. Again we will simplify the real process. We will allocate in our 8-bit storage 6 bits for the *mantissa* (the part to the left of the decimal place) and 2 bits for the fractional portion. We can still have overflow if we try to put a number that is too large for our storage. This storage will hold numbers from 0 to 63.75.

2^5	2^4	2^3	2^2	2^1	2^0	2^{-1}	2^{-2}
32	16	8	4	2	1	1/2	1/4

6. Let's convert 7.25 to our binary representation.

2^5	2^4	2^3	2^2	2^1	2^0	2^{-1}	2^{-2}
0	0	0	1	1	1	0	1

Perfect!

Now try 7.4. The first 6 bits are the same, 000111, but what are we going to do for the decimal places? We have four choices:

00 -> 0
01 -> .25
10 -> .5
11 -> .75

We have no other choices. So it looks like we would have to choose between .25 and .5. This is called round-off error. Converting from decimal numbers with fractional parts to a binary representation can cause errors. We are using a very simplified version here, so the errors would usually occur much further out in the decimal places. Most of the time we would not see these errors but as programmers, we should be aware of them. This is why it is never a good idea to compare two floating numbers using "=". It is much safer to compare the two and check for a certain level of accuracy.

`(Math.abs(x - y) < .000000001)` is better than `(x == y)`

This is also why a programmer should choose the best data type for the job.

7. The third type of error is an under-flow error. Let's use our 8-bit storage scheme again, and try to store the decimal number .1.

2^5	2^4	2^3	2^2	2^1	2^0	2^{-1}	2^{-2}
0	0	0	1	1	1	0	1

The first 6-bits are 000000. Remember the four choices of decimals:

00 -> 0
01 -> .25
10 -> .5
11 -> .75

The number we are trying to store is too small for our storage scheme. Converting from decimal numbers to binary does not work for numbers very close to 0.

Answers to Practice Questions:

10 110 101₂ -> 265₈
3F1₁₆ -> 0011 1111 0001₂
352₈ -> 011 101 010₂
482 -> 0001 1110 0010₂ (Hint: convert to hexadecimal 1st)
482 -> 742₈
482 -> 1E2₁₆
10001₂ -> 17₁₀
5776₈ -> 3070₁₀
3DB₁₆ -> 987₁₀
110 111 010₂ -> 672₈
1011 0010 1111₂ -> B2F₁₆
3FA₁₆ -> 0011 1111 1010₂
712₈ -> 111 001 010₂

Summary/Review

page 10 of 11

There are some dangers and pitfalls in assuming that a computer will always give you the correct answer. Understanding the inner workings of a computer will help you write better programs. Also, hexadecimal numbers are used quite commonly in web design and also in many graphical editing programs.

Chapter 22

A22 Introduction

page 1 of 6

Computers are very complex and interesting machines. Any computer programmer should know as much about them as possible. An understanding of the possible problems associated with computers can be useful to avoid the possible negative results of using them.

The key topics for this lesson are:

Created by mu

- A. [Hardware](#)
- B. [Software](#)
- C. [Networking](#)
- D. [Ethics](#)

A. Hardware

page 2 of 6

1. In order for a computer to make calculations and process jobs, it must have a designated location to store, read and write the information that it is being manipulated. The most integral part of a computer's internal memory is called primary memory, which has generally been in the form of Random Access Memory (commonly known as RAM) since 1968. When a computer is engaged in processing a task, data is stored in the computer's primary memory. A computer's primary memory only holds the data for as long as the computer is powered up with a steady flow of electrical current - it is lost when the computer is shut down.
2. Another form of memory that is very important to the modern computer is called secondary memory. Secondary memory is different from primary memory in the speed of access, the duration of the memory, and the total capacity. It is generally slower than primary memory, which makes it a poor choice for processing jobs. However, secondary memory is usually higher capacity and does not need a constant current running through it to retain its memory, so it is useful for long-term storage. Modern versions of secondary memory storage hardware range from hard disk drives to floppy disk drives to flash drives, as well as to optical discs such as CD-ROMs and DVDs.
3. The Central Processing Unit (CPU) is also a very necessary component of computer hardware. The CPU, known colloquially as the computer's 'brain', processor, or central processor, conducts all of the data calculations and processes for the data held in the computer's memory. The CPU is housed in a silicon chip called a microprocessor. These components understand the basic computer instructions that are provided by the computer's software. Even though CPUs are often physically small, they can contain millions of parts.
4. Storage devices (such as disk drives), display devices (such as computer monitors or screens), output devices (printers, speakers), input devices (keyboards, mice, joysticks, scanners) and other peripherals (modems and network cards) are all considered types of hardware - that is, computer components that can actually be physically touched.

1. Software is a general term for all the computer instructions or data (usually in the form of programs or applications) that exist on a computer. Programs are stored in memory and generally do not have a tangible aspect. Software can range from computer games to word processors (applications software) to the computer's operating system (OS or system software), which manages and controls all the other software on the system.
2. Operating systems have been evolving for many years and have become more and more advanced. The major operating systems competing for the personal computer market today are various versions of Microsoft Windows (i.e., 2000, NT, XP Home, XP Pro), Apple Macintosh (i.e., OS 8, 9, X, Tiger), and Linux (there are dozens of operating systems based on Linux that have their own individual strengths and weaknesses).
3. The main object of this curriculum is to teach students how to create their own computer software. When creating computer software programs as students have already been doing, a language translator and compiler are required to convert the Java code into software 'language' that the CPU is able to understand. Without the aid of these tools, it would be necessary to program computers entirely in binary, with 1's and 0's. Imagine debugging code that looked like that!

1. A network of computers is defined as two or more computer systems connected to each other, in order to communicate and share data. Networks may be a simple linking of computers found close together (in one's home, school, or in the same building), or they may span a relatively large geographical area. A wide-area network (known as a WAN) consists of two or more local-area networks (known as LANs). The largest WAN in existence is the Internet, an interconnected system of networks, which has been expanding rapidly around the world over the last couple of decades. There are millions of websites on the Internet today. This rapid expansion is blurring the definition of networks since a few simple clicks can connect a user to hundreds of different computers within a few seconds.
2. There are many benefits for networking computers together. The most basic benefit is

for sharing and communicating data. The Internet and the World Wide Web have grown exponentially, providing an efficient exchange of information that is unprecedented - a worldwide information highway. Another feature of networking that is becoming popular is called thin client - this exists when a computer (client) in a client-server network has little or no hardware or software on it. The client's computer depends primarily on the centralized and managed host server for processing activities.

D. Ethics

page 5 of 6

1. The power and expansion of computers and networking has unfortunately brought many ethical questions and situations to the forefront. While computers are powerful tools for many aspects of our modern lives and have provided us with amazing advances in many fields from astrophysics to zoology, they have in turn given us many new problematic choices for action and choice. The burgeoning field of computer ethics includes consideration of both personal and social policies for the ethical use of computer technology.
2. Companies have emerged to specialize in the manufacture of anti-virus, anti-spyware, anti-spam, and privacy protection software programs. Nearly everyone has heard of computers getting infected with a malicious computer virus and behaving strangely. Trojan Horses, invasive programs disguised as other things, such as music videos or games, can cause problems with computer systems and compromise a computer user's privacy. Spyware can latch onto a computer and send all sorts of data to third parties, who use this knowledge to adjust their advertising.
3. A whole new field of criminal justice has emerged which deals only with serious computer crimes, such as unauthorized use of a computer (i.e., the theft of usernames and passwords), identity theft, denial of service attacks, the alteration of websites, illegal or offensive material posted on websites or received in emails, predatory sexual behavior, copyright infringement, unregulated gambling, and more. Sexual predators can hide behind a screen name. Computer hackers attempt to get bank account and credit card information. Pornography and gambling can often be hard to regulate over the Internet, and copyright infringement is much more likely to go unnoticed.
4. Since most of these issues and problems are fairly new due to the expansion of the Internet and information technologies everywhere, laws and rules are just starting to be implemented, although it has been a challenge to keep up with the rapid acceleration of changes. It is always important to think about the ramifications of one's own actions

when using computer technology.

Summary/Review

page 6 of 6

It is strongly recommended that computer programmers know how computers function. An understanding of how they work can help to demystify some of the computing processes that are going on behind the scenes as well as to help clarify some of the quirky things that computers do. Understanding the history of computers and their development over time can give us insight as to where they may be heading in the future and help computer programmers to plan for the inevitable changes. The Internet holds a vast amount of information on many subjects related to computers, and should be explored for the depth of content located there.

Chapter 23

AB23 Introduction

page 1 of 9

Two-dimensional arrays allow the programmer to solve problems involving rows and columns. Many data processing problems involve rows and columns, such as an airplane seat reservation system or the mathematical modeling of bacterial growth. A classic problem involving two-dimensional arrays is the bacteria simulation program presented in Lab Assignment AB23.1, *Life*. After surveying the syntax and unique aspects of larger data structures, we will apply them to more challenging lab assignments.

The key topics for this lesson are:

- A. [Two-Dimensional Arrays](#)
- B. [Passing Two-Dimensional Arrays to Methods](#)
- C. [Two-Dimensional Array Algorithms](#)

MATRIX
COLUMN

ROW
LENGTH

A. Two-Dimensional Arrays

page 3 of 9

1. Often the data a program uses comes from a two-dimensional situation. For example, maps are two-dimensional (or more), the layout of a printed page is two-dimensional, a computer-generated image (such as on your computer's screen) is two-dimensional, and so on.

For these situations, a Java programmer can use a *two-dimensional array*. This allows for the creation of table-like data structures with a row and column format. The first subscript is a row index in a table while the second subscript is a column index in a table. Here is example code, in Sample Code 23-1, including a diagram of the array table.

Sample Code 23-1

```
int[][] table = new int[3][4];
int row, col;

for (row = 0; row < 3; row++){
    for (col = 0; col < 4; col++){
        table[row][col] = row + col;
    }
}
```

table

	0	1	2	3
0	0	1	2	3
1	1	2	3	4
2	2	3	4	5

2. Two-dimensional arrays are objects. A variable such as `table` is a reference to a 2D array object. The declaration

```
int[][] table;
```

defines a `table` that holds a reference to a 2D array of integers. Without any further initialization, `table` holds the value `null`.

3. The declaration

```
int[][] table = new int[3][4];
```

defines a `table` that holds a reference to a 2D array of integers, creates an array object of 3 rows and 4 columns, and puts the reference in `table`. All the elements of the array are initialized to zero.

4. The declaration

```
int[][] table = { {0,0,0,0},  
                  {0,0,0,0},  
                  {0,0,0,0} };
```

does exactly the same thing as the previous declaration (and would not ordinarily be used.)

5. The declaration

```
int[][] table = { {0,1,2,3},  
                  {1,2,3,4},  
                  {2,3,4,5} };
```

creates an array of the same dimensions (same number of rows and columns) as the previous array and initializes the elements to the given values in each cell.

6. If no initializer is provided for an array, then when the array is created, it is

automatically filled with the appropriate values: zero for numbers, **false** for **boolean**, and **null** for objects.

7. Just as with one-dimensional arrays, the row and column numbering of a 2-D array begins at subscript zero (0). The 3 rows of the table are numbered from 0...2. Likewise, the 4 columns of the table are numbered from 0...3.
8. The array `table` is filled with the sums of `row` and `col`, which is accomplished by Sample Code 23-2 (see below). To access each location of the matrix, specify the row coordinate first, then the column:

```
table[row][col]
```

Each subscript must have its own square brackets.

9. The length of a 2D array is the number of rows it has. The row index will run from 0 to `length - 1`. The number of rows in `table` is given by the value `table.length`.

Each row of a 2D array has its own length. To get the number of columns in `table`, use any of the following:

```
table[0].length  
table[1].length  
table[2].length.
```

There is actually no rule that says that all the rows of an array must have the same length, and some advanced applications of arrays use varying-sized rows. But if you use the **new** operator to create an array in the manner described above, you'll always get an array with equal-sized rows.

10. The routine that assigned values to the array used the specific numbers of rows and columns. That is fine for this particular program, but a better definition would work for an array of any two dimensions.

Sample Code 23-2

```
int[][] table = new int[3][4];  
int row, col;  
  
for (row = 0; row < table.length; row++){  
    for (col = 0; col < table[row].length; col++){  
        table[row][col] = row + col;  
    }  
}
```

In Sample Code 23-2, the limits of the for loops have been redefined using `table.length` and `table[row].length` so that they work with any two-dimensional

array of ints with any number of rows and columns .

B. Passing Two-Dimensional Arrays to Methods

page 4 of 9

1. The following program in Sample Code 23-3 illustrates parameter passing of an array. The purpose of this method is to print out the array.

Sample Code 23-3

```
// A program to illustrate 2D array parameter passing

public void printTable (int[][] pTable){
    for (int row = 0; row < pTable.length; row++){
        for (int col = 0; col < pTable[row].length; col++){
            System.out.printf("%4d", pTable[row][col]);
        }
        System.out.println();
    }
}
```

2. The `printTable` method uses a reference parameter, `int[][] pTable`. The local identifier `pTable` serves as an alias for the actual parameter `grid` passed to the method.
3. When a program is running and it tries to access an element of an array, the Java virtual machine checks that the array element actually exists. This is called *bounds checking*. If the program tries to access an array element that does not exist, the Java virtual machine will generate an `ArrayIndexOutOfBoundsException` exception. Ordinarily, this will halt the program.

C. Two-Dimensional Array Algorithms

page 5 of 9

1. The most common 2-D array algorithms will involve processing the entire grid, usually row-by-row or column-by-column.
2. Problem-solving on a matrix could involve processing:

- a. one row
- b. one column
- c. one cell
- d. adjacent cells in different directions

Summary/Review

page 6 of 9

Two-dimensional arrays will be applied to two interesting problems. The simulation of life in a petri dish of bacteria will require a two-dimensional array representation. The second and third lab assignments are different versions of the “Knight’s Tour” problem - an interesting and demanding chess movement problem.

Chapter 24

AB24 Introduction

page 1 of 7

Suppose that you are trapped inside a maze and you need to find your way out. Initial attempts at exiting are random and because the walls all appear identical, you begin wasting time and energy trying the same routes over and over again. After some thought and after recalling a useful computer science algorithm, you take advantage of a tool you have in your pocket - a marking pen. You begin marking the walls with some notes and a time stamp. (This technique recalls the one used by Hansel and Gretel in the Grimm fairy tale, when Hansel marked his path through the woods with shiny stones (and then breadcrumbs), so that he could safely find his way back.) At each branch point, you mark the direction you are about to attempt with an arrow. If a dead-end is encountered just around the corner, you retreat to the branch point, mark “dead-end” for that direction, and try another direction. By marking the directions that have failed, you avoid needlessly trying them again.

This two-dimensional problem involving backtracking provides us with a recursive matrix problem (and its entire solution), translated from *Oh! Pascal!*, 2nd edition. Your programming assignment will also be a recursive matrix problem.

The key topics for this lesson are:

- A. [Defining the Maze Problem](#)
- B. [Developing the Recursive Solution](#)
- C. [The Solution to the Maze Problem](#)

[AB24 Vocabulary](#)

page 2 of 7

BACKTRACKING

A. Defining the Maze Problem

page 3 of 7

1. The maze will be defined as a 2-D grid of asterisks (marking walls) and blanks (marking potential paths). The size of the grid will be 12 x 12 and the starting location will be at 6, 6.
2. The data structure for this problem will be a 2-dimensional array of `char`. The array will be declared as a 13 x 13 grid so that we can use locations 1..12 by 1..12. Row 0 and column 0 will not be used in the data structure.
3. The initial grid is stored as a text file and will be loaded into the array.

(Here is a copy of the *mazeData.txt* file)

```

*** *****
***      *****
***  **  *****
***      *    ***
***** **  ***
***** **  ***
***** *****
*****      *****
*****      *    ***
*      **  *****
* *      *****
* *****

```

4. The solution is a backtracking algorithm involving a series of trial and error attempts. A potential path out of the maze will be traced with a !, with moves limited to horizontal or vertical directions. If a dead-end is encountered, back up and try a different direction.
5. We will be at an exit point if we arrive at row 1 or MAXROW, or at column 1 or MAXCOL.

B. Developing the Recursive Solution

page 4 of 7

1. In developing a recursive solution, consider the base cases first. What situation(s) cause the recursive method to exit?
 - a. We have arrived at a location that is off the data structure. It is important to catch this situation first to avoid an array indexing error.
 - b. Encountering a '*' character means that we have run into a wall. The algorithm should stop.
 - c. Encountering a location we have already visited should cause the algorithm to stop.
 - d. Arriving at a location that has a row or column value equal to 1 or MAXROW or MAXCOL means that we have found an exit point.
2. The general case of encountering a blank space requires the following steps:
 - a. Change the character value from the blank space to the '!' character.

- b. Check to see if you are at an exit point. If so, print the maze with a trail of '!' markers to the exit point.
 - c. If you are not at an exit point, make 4 recursive calls to check all 4 directions, feeding the new coordinates of the 4 neighboring cells.
3. When an exit point is found, print only the successful trail of '!' marks that leads to an exit. As a result, it is necessary for each recursive call to work with a copy of the array. Why is this? If a reference to the original array is passed with each recursive call, the placement of '!' marks would be permanent in the data structure.

C. The Solution to the Maze Problem

page 5 of 7

```
import java.io.File;
import java.util.Scanner;
import java.io.IOException;

/**
 * Description of the Class
 *
 * @author Administrator
 * @created July 16, 2002
 */
class ThreadTheMaze{
    /**
     * Description of the Field
     */
    private final static char BLANK = ' ';
    private static final int MAXROW = 12;
    private static final int MAXCOL = 12;
    private int myMaxRow;
    private int myMaxCol;
    private char [][] myMaze;

    public ThreadTheMaze(){
        myMaze = new char [MAXROW + 1][MAXCOL + 1];
        myMaxRow = myMaze.length - 1;
        myMaxCol = myMaze[0].length - 1;
    }

    /**
     * Initiates the trace process
     *
     * @param none
     */
}
```

```

public void doTraceMaze() {
    loadMaze();
    traceMaze(myMaze, myMaxRow/2, myMaxCol/2);
}

/**
 * Loads the maze characters from mazeData.txt
 *
 * @param maze Description of Parameter
 */
private void loadMaze(){
    String line;
    Scanner in;
    try{
        in = new Scanner(new File("mazeData.txt"));

        for (int row = 1; row <= myMaxRow; row++){
            line = in.nextLine();
            for (int col = 1; col <= myMaxCol; col++){
                myMaze[row][col] = line.charAt(col-1);
            }
        }
    }catch(IOException i){
        System.out.println("Error: " + i.getMessage());
    }
}

/**
 * Description of the Method
 *
 * @param maze Description of Parameter
 */
public void printMaze(char[][] maze){
    Scanner console = new Scanner(System.in);

    for (int row = 1; row <= myMaxRow; row++){
        for (int col = 1; col <= myMaxCol; col++){
            System.out.print("" + maze[row][col]);
        }
        System.out.println();
    }
    System.out.println();
    System.out.println("Hit enter to see if there are more
solutions.");
    String anything = console.nextLine();
}

/**
 * Will attempt to find a path out of the maze. A
 * path will be marked with the ! marker. The method
 * makes a copy of the array each time so that only
 * the path out will be marked, otherwise extra !
 * markers will appear in the answer.
 * The function is recursive.

```

```

*
* @param maze Description of Parameter
* @param row Description of Parameter
* @param col Description of Parameter
*/
public void traceMaze(char[][] maze, int row, int col){
    //char[][] mazeCopy = (char[][])maze.clone();

    char[][] mazeCopy = new char[maze.length][maze[0].length];
    for (int r = 0; r < mazeCopy.length; r++){
        for (int c = 0; c < mazeCopy[0].length; c++){
            mazeCopy[r][c] = maze[r][c];
        }
    }

    if (1 <= row && row <= myMaxRow && 1 <= col && col <= myMaxCol){
        // boundary check, if false, a base case
        if (BLANK == mazeCopy[row][col]){
            // if false, base case
            mazeCopy[row][col] = '!';
            if (1 == row || myMaxRow == row || 1 == col || myMaxCol ==
col){
                printMaze(mazeCopy); // base case
            }else{
                traceMaze(mazeCopy, row - 1, col);
                traceMaze(mazeCopy, row, col + 1);
                traceMaze(mazeCopy, row + 1, col);
                traceMaze(mazeCopy, row, col - 1);
            }
        }
    }
}
}

```

Here are the two solutions when starting at location (6,6):

<pre> *** !***** *** ! ***** *** !** ***** *** !!* *** ***** !** *** ***** !** *** ***** ***** ***** ***** ***** * ***** * ** ***** * * ***** * ***** </pre>	<pre> *** ***** *** ***** *** ** ***** ***** ** *** ***** ** *** ***** !** *** ***** !***** ***** !!* ***** ***** * !***** * !!*!!*!!***** * !*!!*!!***** * !***** </pre>
---	---

1. It is very significant that a blank space be changed to an '!' mark before the recursive calls are made. For example, suppose you began at location 6,6 and a blank space

was encountered. If you didn't change the blank to an '!' mark, the recursive call to solve the upward direction would receive the location 5,6. This recursive call would eventually look back down at location 6,6 - the location where it came from. A blank space would still be there and the recursive calls would go around in circles until the computer ran out of memory.

2. Changing the blank space to an '!' mark is like leaving a trail of markers. When a recursive call of a neighboring cell looks back at the cell from which it came, it will see a '!' mark and not a blank space.

Some of the earlier examples of recursion were applied to linear problems. The factorial or Fibonacci problems were one dimensional, which required only one recursive call within the algorithm. The maze problem is best-solved recursively because of the different directions that the problem solving takes. Loosely stated, any problem that involves multiple directions and returning to a branch point is a likely candidate for recursion.

Chapter 25

Two critical criteria used for selecting a data structure and algorithm are the amount of memory required and the speed of execution. The analysis of the speed of an algorithm leads to a summary statement called the order of an algorithm.

The key topics for this lesson are:

- A. [Order of Algorithms](#)
- B. [Constant Algorithms, \$O\(1\)\$](#)
- C. [log₂N Algorithms, \$O\(\log_2 N\)\$](#)

- D. [Linear Algorithms, \$O\(N\)\$](#)
- E. [N * \$\log_2 N\$ Algorithms, \$O\(N * \log_2 N\)\$](#)
- F. [Quadratic Algorithms, \$\(N^2\)\$](#)
- G. [Other Orders](#)
- H. [Comparison of Orders of Algorithms](#)

[AB25 Vocabulary](#)

page 2 of 11

BIG O NOTATION

CUBIC ORDER

LOG₂ N ORDER

ORDER OF ALGORITHM

CONSTANT ORDER

LINEAR ORDER

N LOG₂ N ORDER

QUADRATIC ORDER

A. Order of Algorithms

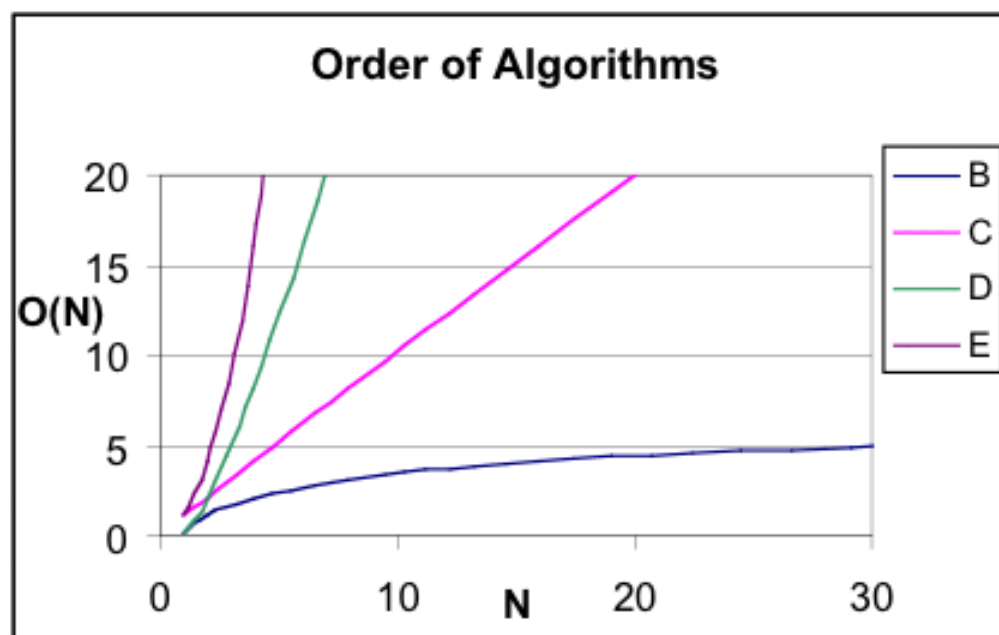
page 3 of 11

1. The order of an algorithm is based on the number of steps that it takes to complete a task. Time is not a valid measuring stick because computers have different processing speeds. We want a method of comparing algorithms that is independent of the computing environment and the speed of the microprocessor.
2. Most algorithms solve problems involving an amount of data, N. The order of algorithms will be expressed as a function of N, the size of the data set.
3. The following chart summarizes the numerical relationships of common functions of N.

A N	B $O(\log_2 N)$	C $O(N)$	D $O(N * \log_2 N)$	E $O(N^2)$
1	0	1	0	1
2	1	2	2	4
4	2	4	8	16

8	3	8	24	64
16	4	16	64	256
32	5	32	160	1024
64	6	64	384	4096
128	7	128	896	16384
256	8	256	2048	65536
512	9	512	4608	262144
1024	10	1024	10240	1048576

- The first column, N, is the number of items in a data set.
- The other four columns are mathematical functions based on the size of N. In computer science, we write this with a capital O (order) instead of the traditional F (function) of mathematics. This type of notation is the order of an algorithm, or Big O notation.
- The graph below, Order of Algorithms, gives a clearer sense of the relationships among the columns of numbers. Since the vertical axis represents the theoretical number of steps required by an algorithm to sort a list of N items, lines B and C represent more efficient algorithms than D and E. Today's data sets can grow to enormous sizes, so algorithm designers are always searching for ways to reduce the number of steps, even on the fastest supercomputers.



- You have already seen column E in an experimental sense when you counted the number of steps in the quadratic sorting algorithms. The relationship between columns A and E is quadratic - as the value of N increases, the other column

increases as a function of N^2 . The graph of column E is a portion of a parabola.

B. Constant Algorithms, $O(1)$

page 4 of 11

1. This relationship was not included in the *Order of Algorithms* graph. Here, the size of the data set does not affect the number of steps this type of algorithm takes. For example:

```
public int howBig(ArrayList <Integer> list){  
    return list.size();  
}
```

2. The number of data items in the array could vary from 0...4000, but this does not affect the `howBig` algorithm. It will take one step regardless of how big the data set is.
3. A constant time algorithm could have more than just one step, as long as the number of steps is independent of the size (N) of the data set.

C. $\log_2 N$ Algorithms, $O(\log_2 N)$

page 5 of 11

1. A logarithm is the exponent to which a base number must be raised to equal a given number.
2. A $\log_2 N$ algorithm, seen in line B on the *Order of Algorithms* graph, is one where the number of steps increases as a function of $\log_2 N$. If the number of data was 1024, the number of steps equals $\log_2 1024$, or 10 steps since $2^{10} = 1024$.
3. Algorithms in this category involve splitting the data set in half repeatedly. Several examples will be encountered later in the course.
4. Algorithms that fit in this category are classed as $O(\log N)$, regardless of the numerical base used in the analysis.

1. This is an algorithm where the number of steps is directly proportional to the size of the data set, seen in line C on the *Order of Algorithms* graph. As N increases, the number of steps also increases.

```
public long sumData(ArrayList <Integer> list){
    long total = 0;

    Iterator <Integer> itr = list.iterator();
    while(itr.hasNext()){
        total += itr.next();
    }

    return total;
}
```

2. In the above example, as the size of the array increases, the number of steps increases at the same rate.
3. A non-recursive linear algorithm, $O(N)$, always has a loop involved.
4. Recursive algorithms, in which the looping concept is developed through recursive calls, are usually linear. For example, the recursive factorial function is a linear function.

```
public long fact (int n) {
// precondition: n > 0

    if (1 == n)
        return 1;
    else
        return n * fact(n - 1);
}
```

The number of calls of fact will be n . Inside of the function is one basic step, an **if/else**. So we are executing one statement n times.

1. Algorithms of this type, seen in line D on the *Order of Algorithms* graph, have a $\log_2 N$ procedure that must be applied N times.
2. The recursive MergeSort is a $O(N * \log_2 N)$ algorithm.
3. As the graph shows, these algorithms are markedly more efficient than our next category, quadratics.

F. Quadratic Algorithms, (N^2)

page 8 of 11

1. This is an algorithm, seen in line E on the *Order of Algorithms* graph, in which the number of steps required to solve a problem increases as a function of N^2 . For example, here is bubbleSort.

```
void bubbleSort(ArrayList <Comparable> list){
    for (int outer = 0; outer < list.length - 1; outer++){
        for (int inner = 0; inner < list.size()-outer-1; inner++){
            if (list.get(inner).compareTo(list.get(inner + 1)) > 0){
                //swap list[inner] & list[inner+1]
                Comparable temp = list.get(inner);
                list.set(inner, list.get(inner + 1));
                list.set(inner + 1, temp);
            }
        }
    }
}
```

2. The **if** statement is buried inside nested loops, each of which is tied to the size of the data set, N . The **if** statement is going to be executed approximately N times for each of N items, or N^2 times in all.
3. The efficiency of this bubble sort was slightly improved by having the inner loop decrease, but we still categorize this as a quadratic algorithm.
4. For example, the number of times the inner loop happens varies from 1 to $(N-1)$. On average, the inner loop occurs $(N/2)$ times.
5. The outer loop happens $(N-1)$ times, or rounded off N times.
6. The number of times the **if** statement is executed is equal to this expression:

if statements = (Outer loop) * (Inner loop)

if statements = $(N) * (N/2)$

if statements = $N^2/2$

7. The coefficient $1/2$ becomes insignificant for large values of N , so we have an algorithm that is quadratic in nature.
8. When determining the order of an algorithm, we are only concerned with its category, not a detailed analysis of the number of steps.

G. Other Orders

page 9 of 11

1. A cubic algorithm is one where the number of steps increases as a cube of N , or N^3 .
2. An exponential algorithm is one where the number of steps increases as the power of a base, like 2^N .
3. Both of these categories are astronomical in the number of steps required. Such algorithms are avoided when possible, and for large values of N can run slowly on even the fastest computers.

H. Comparison of Orders of Algorithms

page 10 of 11

1. We obviously want to use the most efficient algorithm in our programs. Whenever possible, choose an algorithm that requires the fewest number of steps to process data.

When designing solutions to programming problems, we are often concerned about finding the most efficient solutions regarding time and space. We will consider memory requirements at a later time. Speed issues are based on the number of steps required by algorithms.

Chapter 26

AB26 Introduction

page 1 of 6

Quicksort is another recursive sorting algorithm that works by dividing lists in half. Whereas mergeSort divided lists in half and then sorted each sublist, QuickSort will roughly sort the entire list, and then split the list in half. The order of these two sorts falls into the category $O(N * \log N)$, which was introduced in Lesson AB25, *Order of Algorithms*. When the lists become large, either of these sorts will do an excellent job.

The key topics for this lesson are:

- A. [QuickSort](#)
- B. [Order of QuickSort](#)

[AB26 Vocabulary](#)

page 2 of 6

QUICKSORT

1. Here is the overall strategy of QuickSort:

- a. QuickSort chooses an arbitrary value from somewhere in the list. A common location is the middle value of the list.
- b. This value becomes a decision point for roughly sorting the list into two sublists, which are called the “left” and the “right” sublists. All the values smaller than the dividing value are placed in the left sublist, while all the values greater than the dividing value are placed in the right sublist.
- c. Each sublist is then recursively sorted with QuickSort.
- d. The termination of QuickSort occurs when a list of one value is obtained, which by definition is sorted.

2. This is the code for quickSort:

```

void quickSort (ArrayList <Comparable> list, int first, int last){
    int g = first, h = last;

    int midIndex = (first + last) / 2;
    Comparable dividingValue = list.get(midIndex);
    do{
        while (list.get(g).compareTo(dividingValue) < 0) g++;
        while (list.get(h).compareTo(dividingValue) > 0) h--;
        if (g <= h){
            //swap(list[g], list[h]);
            swap(list,g,h);
            g++;
            h--;
        }
    }while (g < h);
    if (h > first) quickSort (list,first,h);
    if (g < last) quickSort (list,g,last);
}

```

3. Suppose we have the following list of 9 unsorted values in an array:

17 22 34 28 19 84 11 92 1

- a. The values received by the formal parameter `list` in the first call of `quickSort` are a reference to an array, and the first and last cells of the array, 0 and 8. Variable `first` is initialized with the value 0, and `last` is initialized with the value

8.

- b. The `midIndex` value is calculated as $(0+8) / 2$, which equals 4.
- c. The `dividingValue` is the value in location 4, `list[4] = 19`.
- d. The value of 19 will be our decision point for sorting values into two lists. The list to the left will contain all the values less than or equal to 19. The list to the right will contain the values larger than 19. Or simply put, small values go to the left and large values go to the right.
- e. The identifiers `g` and `h` will be indices to locations in the list. The **while** loops will move `g` and `h` until a value is found to be on the wrong side of the dividing value of 19. The `g` index is initialized with the value of `first` and the `h` index is initialized with the value of `last`.
- f. The `g` index starts at position 0 and moves until it “sees” that 22 is on the wrong side. Index `g` stops at location 1.
- g. The `h` index starts at position 8. Immediately it “sees” that the value 1 is on the wrong side.
- h. Since $g \leq h$ ($1 \leq 8$), the values in `list[1]` and `list[8]` are swapped. After the values are swapped, index `g` moves one position to the right and index `h` moves one position to the left.
- i. The values of the pointers are now: `g = 2`, `h = 7`. We continue the **do-while** loop until `g` and `h` have passed each other, that is when $g > h$. At this point, the lists will be roughly sorted, with values smaller than 19 on the left, and values greater than 19 on the right.
- j. If the left sublist has more than one value, (which is determined by the $h > \text{first}$ expression), then a recursive call of `quickSort` is made. This call of `quickSort` will send the index positions that define that smaller sublist.
- k. Likewise, if the right sublist has more than one value, `quickSort` is called again and the index positions that define that sublist are passed.

1. Determining the order of QuickSort, $O(N \log_2 N)$, is a difficult process. The best way to understand it is to imagine a hypothetical situation in which each call of `quickSort` results in sublists of the same size. Let's try a size of 128, because it is a power of 2.
2. If a list has 128 elements, the number of calls of `quickSort` required to move a value into its correct spot is $\log_2 128$, which equals 7 steps. Dividing the list in half gives us the $\log_2 N$ aspect of QuickSort.
3. But we need to do this to 128 numbers, so the approximate number of steps to sort 128 numbers will be $128 * \log_2 128$. A general expression of the order of QuickSort will be $O(N * \log_2 N)$. An $O(N * \log_2 N)$ algorithm is a more specific designation of the broader category called $O(N * \log N)$.
4. A graph of an $O(N * \log_2 N)$ algorithm is close to a linear algorithm, for large values of N . The $\log_2 N$ number of steps grows very slowly, making QuickSort a dramatic improvement over the $O(N^2)$ sorts.

QuickSort is generally the fastest and therefore most widely used sorting algorithm. There is a variation of QuickSort named "QuickerSort" but it is still in the same class of algorithms. Once again, recursion makes fast work of a difficult task.

Chapter 27

`ArrayList`, part of the Java List Collection, was introduced in Lesson A15, *ArrayList*. As demonstrated throughout this curriculum, lists are very useful. This lesson will cover the List Interface and the two different implementations of Lists: `ArrayList` and `LinkedList`. Iterators

will be covered as well as the different methods of storage used by the two implementations.

The key topics for this lesson are:

- A. [The List Interface](#)
- B. [The ArrayList Class](#)
- C. [The LinkedList Class](#)
- D. [Traversing a List using Iterator or ListIterator Objects](#)
- E. [LinkedList VS ArrayList](#)

[AB27 Vocabulary](#)

page 2 of 9

ITERATOR

ListIterator

LINKEDLIST

TRAVERSE

A. The List Interface

page 3 of 9

The AP subset of the Java List interface is shown below:

interface java.util.List - AP subset

```
int size()
boolean add(Object x)
Object get(int index)
Object set(int index, Object x)
Iterator iterator()
ListIterator listIterator()
```

In order to implement a List, you must use the methods listed in the interface. Java provides two implementations for a List: ArrayList and LinkedList.

Like ArrayList and other collection classes, LinkedList stores references to objects.

```
List < ClassName > classList = new LinkedList < ClassName >();
```

Created by mu

```
classList.add(new ClassName(_APCS_));  
...  
ClassName favoriteClass = classList.get(1)
```

This creates `classList` as a `List` implemented as a `LinkedList`. Declaring `classList` as a `List` restricts the methods for `classList` to those in the `List` interface. Instantiating `classList` as a `LinkedList` means it is stored as a `LinkedList`.

B. The ArrayList Class

page 4 of 9

1. The `ArrayList` class, `java.util.ArrayList`, provides the following methods in addition to the `List` methods:

```
void add(int index, Object x)  
Object remove(int index)
```

2. The basics of using the `ArrayList` class were explained in Lesson A15, *ArrayList*.

C. The LinkedList Class

page 5 of 9

1. The `LinkedList` class is based on the linked list data structure, just as the `ArrayList` class is based on the array data structure. An array is a random access data structure of contiguous elements. Any element in an array can be accessed directly using an index. In a linked list, the elements are connected by links and are not necessarily contiguous. In a linked list, the elements cannot be accessed directly; the elements must be accessed sequentially, following the links until the desired element is reached. A good analogy for an array is the viewing of a DVD (digital video disc). Any scene in a movie on a DVD can be accessed directly using the DVD scene selection menu to 'jump' to the scene.

By contrast, playing a videocassette tape is an example of a linked list. To reach a scene with a videocassette recorder/player (VCR), the tape must be advanced through all of the previous scenes. Some high-end VCRs have a feature that automatically goes back and marks the beginning and end of each commercial of a recorded program. Then, when playing the tape — with the commercial advance

feature turned on — the VCR fast-forwards over the ads to skip them. To the user, it appears that the VCR is accessing the marked positions directly. However, the VCR internally sequences through the taped commercials to reach the marked positions. This is similar to what happens in a linked list when accessing an element by an index.

2. The `LinkedList` class, `java.util.LinkedList`, provides the following methods in addition to the `List` methods:

```
void addFirst( Object x)
void addLast(Object x)
Object getFirst()
Object getLast()
Object removeFirst()
Object removeLast()
```

3. To declare a reference variable for a `LinkedList`, do the following.

```
// myLinkedList is a reference to an LinkedList
// object.
LinkedList <ClassName> myLinkedList =
new LinkedList <ClassName> ();
```

4. An object can be declared as a `List` and created as a `LinkedList` or `ArrayList`. In that case, only the methods of `List` are available to the object. The advantage is that either implementation can be chosen without having to change any of the code that uses the object.

```
if (implementationChoice.equals("LinkedList")) {
    List <String> list = new LinkedList <String>();
}
if (implementationChoice.equals("ArrayList")) {
    List <String> list = new ArrayList <String>();
}
list.add("John");
list.add("George");

for(String temp : list){
    System.out.println(temp);
}
```

Notice that the for each loop can be used here. A for each loop can be used in any collection that implements `Iterable`.

1. A traversal of a list is an operation that visits all the elements of the list in sequence and performs some operation.
2. An iterator is an object associated with the list. When an iterator is created, it points to a specific element in the list, usually the first. The iterator provides methods to check whether there are more elements to be visited, to obtain the next element, and to remove the current element. This makes it easy to traverse a list. Using an iterator helps prevent OBOBs ('Off-By-One-Bugs': attempts to access an element beyond the length of the list) because the iterator knows how big the list is.
3. In Java, iterators are defined by the library interface, `java.util.Iterator`.
4. Iterator defines three basic methods:

```
Object next()  
// Returns the next element in the iteration  
  
boolean hasNext()  
// Returns true if the iteration has more elements  
  
void remove()  
// Removes the last element returned by next from the list
```

5. A list traversal would be implemented using an iterator as follows.

```
LinkedList <String> list = new LinkedList <String>();  
// Add values to the list  
...  
  
Iterator <String> iter = list.iterator();  
while (iter.hasNext()){  
    System.out.println(iter.next());  
}
```

Note that the list provides the iterator.

6. A limitation of the `Iterator` interface is that an iterator always starts at the beginning of the list and can only move forward.
7. A more comprehensive `ListIterator` object is returned by `List`'s `listIterator` method. `ListIterator` extends `Iterator`. A `ListIterator` can start at any specified position in the list and can proceed forward or backward. For example:

```
ListIterator <String> listIter =  
list.listIterator(list.size());  
while (listIter.hasPrevious()){  
    String value = listIter.previous();  
    // process value
```

```
    ...  
}
```

8. Some useful `ListIterator` methods are summarized below:

```
Object next()  
// Returns the next element in the iteration.  
  
void add(Object x)  
// Inserts the argument into the list being iterated over.  
  
void set(Object x)  
// Replaces the last element returned by next by the argument.
```

9. The following two examples illustrate one of the advantages of iterators.

Example 1:

```
LinkedList <String> list = new LinkedList <String>();  
// Add values to the list  
  
list.add("John");  
list.add("Ann");  
list.add("Betsy");  
list.add("Nancy");  
  
for(int i = 0; i < list.size(); i++){  
    String value = list.get(i);  
    System.out.println(value);  
    if(value.equals("Ann"))  
        list.remove(i);  
}
```

Run Output:

```
John  
Ann  
Nancy
```

Example 2:

```
LinkedList <String> list2 = new LinkedList <String>();  
// Add values to the list  
  
list2.add("John");  
list2.add("Ann");  
list2.add("Betsy");  
list2.add("Nancy");  
Iterator <String> listIter = list2.iterator();  
while (listIter.hasNext()){  
    String value = listIter.next();  
    System.out.println(value);  
}
```

```

        if(value.equals("Ann"))
            listIter.remove();
    }

```

Run Output:

```

John
Ann
Betsy
Nancy

```

In Example 1, the index is external to the list, so after removing Ann, i is incremented but the elements of the list have changed positions and Betsy is skipped. In Example 2, the iterator keeps track of the positions of the elements after Ann is removed and Betsy is not skipped.

E. LinkedList vs ArrayList

page 7 of 9

An important aspect of good programming is to choose the best data structure to solve a particular problem. Here are some advantages and disadvantages of using one versus the other — LinkedList VS ArrayList:

Operation	LinkedList	ArrayList
Searching	Best performance is $O(N)$ because the list must be searched sequentially even if it is ordered.	Best performance is $O(\log N)$ if the list is ordered because a binary search can be used.
Insertion/deletion	Performance is $O(1)$ no matter how big the list is. At most, two elements are affected (see Lesson AB30).	Performance is $O(N)$ because many elements of the list may have to be moved.
Accessing by index	Performance is $O(N)$.	Performance is $O(1)$.

When designing a program to build a telephone directory, for example, ArrayList might be a better choice than LinkedList. The directory would not change often, so insertion/deletion would not be a major concern. However, fast searching would be important.

On the other hand, LinkedList might be better for a print spooler program. Print jobs will

frequently be inserted (when a new print request is received) and deleted (when the print job is sent to the physical printer). The list could even be prioritized so that small print jobs would be inserted near the front of the list. However, searching through the list for a particular print job would not be a common occurrence.

Java provides two implementations of the `List` interface: `ArrayList` and `LinkedList`. Both implementations allow a program to use the standard methods of `List` as well as to take advantage of methods specific to each implementation. Iterators are useful for traversing lists. Each implementation of `List` has advantages that make it well suited for different tasks.

Chapter 28

As we saw in Lesson AB27, *Java Lists and Iterators*, a collection is an object that holds other objects. It can be used in many situations, but a collection is most often used to add, remove, and otherwise manage the elements in it.

Collections can be implemented in several ways. That is, the data structure that stores the objects can be implemented using different techniques. In this lesson, Sets and Maps will be studied. A Set is a collection of elements without duplicates. A Map is a group of elements that can be referenced by a key value.

The key topics for this lesson are:

- A. [Sets](#)
- B. [TreeSet](#)
- C. [HashSet](#)

- D. [HashSet VS. TreeSet](#)
- E. [Maps](#)
- F. [TreeMap](#)
- G. [HashMap](#)

[AB28 Vocabulary](#)

page 2 of 11

HASHMAP
KEY
SET
TREEMAP

HASHSET
MAP
TREESET

A. Sets

page 3 of 11

1. In the Java standard class library, the `Set` interface defines the operations on an object that represents a set of elements. The `Set` interface is based on the idea of a mathematical set. A set is a collection that has no duplicate elements.
2. A set must not have two elements that are equal as specified by an object's `equals` method. For example `{7, 11, 13}` is a set but `{7, 11, 11}` is not.
3. The `Set` interface allows the following operations:
 - insert an element into the set
 - remove an element from the set
 - test if a given element is in the set
 - iterate over the elements of the set using `Iterator`
4. Figure 28.1 below lists the methods of the `Set` interface that are a part of the AP subset.

```
// Adds the specified element to this set if it is
// not already present and returns true. If this set
// already contains the specified element, the call
```

Created by mu

```

// leaves this set unchanged and returns false.
boolean add(Object obj);

// Returns true if the set contains the specified
// element, false otherwise.
boolean contains(Object obj);

// Removes the specified element from this set if it
// is present and returns true. Returns false if the
// element was not in the set.
boolean remove(Object obj);

// Returns the number of elements in this set.
int size();

// Returns an iterator over the elements in the set.
Iterator iterator()

```

Figure 28.1 - Methods of the set interface included in the AP Subset

5. Because a set may not have duplicates, the elements of a set should be immutable (unchangeable) objects. If a set contained changeable objects, an object could be changed to become equal to another object in the set.
6. Because the elements in a set do not have to be in order, the set interface does not require that the elements in the iterator returned by the `iterator` method be in any particular order. The order of the values generated by the iterator depends on the class that implements set.

B. TreeSet

page 4 of 11

1. The `java.util.TreeSet` class in the Java standard class library is an implementation of the set interface using trees. This class uses a balanced binary search tree to keep elements in sorted order. The `TreeSet` class implements the set interface, so Figure 28.1 describes `TreeSet`'s methods as well.
2. In general, any `Comparable` objects may be placed into a `TreeSet`. This guarantees that the set will be in ascending order, as determined by the object's `compareTo` method. Items in a `TreeSet` should also be mutually comparable, meaning that for any pair of elements `e1` and `e2` in the set, `e1.compareTo(e2)` will not throw a `ClassCastException`. The items in a `TreeSet` should all be of the same type.

3. Because of the balanced binary tree implementation, the `TreeSet` class provides an $O(\log n)$ run time for the operations `add`, `remove`, and `contains`.

```
import java.util.TreeSet;
import java.util.Set;

Set <String> myTree = new TreeSet <String>();
myTree.add("Nancy");
myTree.add("David");
myTree.add("David");
System.out.println(myTree.size());

for(String temp : myTree){
    System.out.println(temp);
}
```

The output for this code fragment is

```
The size of the set is 2
David
Nancy
```

C. HashSet

page 5 of 11

1. The `java.util.HashSet` class in the Java standard class library is an implementation of the `Set` interface using a hash key. A hash key is an index that is calculated from a value by a hashing algorithm (see Lesson AB32, *Hash-Coded Data Storage* for more information on hash keys). The `HashSet` class implements the `Set` interface; therefore, Figure 28.1 describes `HashSet`'s methods as well.

```
Set <String> s = new HashSet <String>();
s.add("Lynn");
s.add("Nancy");
s.add("David");
s.add("Lynn");
System.out.println("Size of set = " + s.size());
Iterator itr = s.iterator();
while(itr.hasNext()){
    System.out.println(itr.next());
}
```

Run Output:

```
Lynn
David
Nancy
```

2. Since a HashSet does not guarantee any particular order, the order of the output is not known. All we do know is that we will get all of the elements in the Set.
3. Hashing is a way for the data to be stored in an array in such a way that it can be retrieved very efficiently. Hashing is $O(1)$ for add, remove, and contains.

D. HashSet vs. TreeSet

page 6 of 11

Here are some advantages and disadvantages of using a hash compared to using a tree.

Hashing is very efficient as long as the hashing algorithm is a good one. It gives very fast access to any element in the HashSet. However, dealing with the complexities of creating the hashing algorithm as well as dealing with collisions (a collision is when two different items have the same hash key) can be difficult. One consequence of collisions is that extra memory is required to handle the possibility of collisions. Lesson AB32, *Hash-Coded Data Storage* will explore hashing in detail.

Balanced binary search trees give good average performance for any data size. In certain situations, they can be incredibly efficient and always use the minimum amount of memory. Lesson AB30, *Binary Trees* will explore balanced binary search trees in detail.

E. Maps

page 7 of 11

1. A map maintains a correspondence between elements of two sets of objects. The objects in one set are thought to be “keys,” and the objects in the other set are thought to be “values.” A map cannot contain duplicate keys, which means that each key maps to only one value. Different keys, however, can map to the same value.
2. An example of a map is an email directory: the keys in the map are the email addresses and the values are the names in the address book. Each email address in the directory maps to one name. Duplicate keys are not allowed, so the keys in a map form a set. Note that this does not preclude two different keys mapping to the same

value, just as two email addresses can belong to one name. Like the elements of a set, the keys in a Map should be immutable.

3. The Map interface allows the following operations:

- insert a key/value pair into the map
- retrieve any value, given its key
- test if a given key is in the map
- view the elements in the map
- iterate over the mapping elements, using `Iterator`

4. In Java, this functionality is formalized by the `java.util.Map` interface. A few of Map's commonly used methods that are included as part of the AP Subset are shown in Figure 28.2 below.

- a. The `put` method adds a key/value association to the map. If the key was previously associated with a different value, the old association is broken and `put` returns the value previously associated with the key. If the key had no prior association with a value, `put` returns `null`.
- b. The `get` method returns the value associated with a given key or `null` if the key is not associated with any value.
- c. The `containsKey` method returns `true` if a given key is associated with a value and `false` otherwise.

5. Figure 28.2 below lists the methods of the Map interface that are a part of the AP subset.

```
// Adds the key-value pair to this map. Returns the
// previous value associated with specified key, or
// null if there was no mapping for key.
Object put(Object key, Object value);
```

```
// Returns the value to which the specified key is
// mapped, or null if the map contains no mapping
// for this key.
Object get(Object key);
```

```
// Removes the mapping for this key from this map
// if present. Returns the value to which the map
// previously associated with the key, or null if
// the map contained no mapping for this key. Note
// that a return value of null may also indicate
// that this key was previously mapped to null
Object remove(Object key);
```

```
// Returns true if this map contains a mapping for
// the specified key, false otherwise.
boolean containsKey(Object key);

// Returns the number of key-value pairs in the map.
int size();

// Returns a set containing the keys in this map.
Set keySet()
```

Figure 28.2 - Methods of the Map interface included in the AP Subset

F. TreeMap

page 8 of 11

1. The `java.util.TreeMap` class implements the `Map` interface using a balanced binary search tree ordered by keys. In general, any `Comparable` objects may be placed into a `TreeMap` as the key. This guarantees that the keys will be in ascending order, as determined by the key object's `compareTo` method.
2. Because of the balanced binary tree implementation, the `TreeMap` class provides an $O(\log n)$ run time for the operations `put`, `get`, and `containsKey`.
3. The following code fragment demonstrates implementation of a `TreeMap`:

```
TreeMap studentMap = new TreeMap();

for (int recNum = 1; recNum <= NUM_STUDENTS; recNum++){
    Student s = new Student();    // declare a new Student
    s.setName(...);               // set the student's
                                // attributes
    s.setGradeLevel(...);         // ...
    s.setID(...);                 // ...

    studentMap.put(s.getID(), s); // add student to the map
                                // using ID as the key
}
// Display the student whose key is S964413
System.out.println(studentMap.get("S964413"));
```

4. The `Map` interface does not specify a method for obtaining an iterator, and the `TreeMap` class does not have one. Instead, you can get the set of all keys by calling the `keySet` method, then iterate over that set to get all values. For example:

```
TreeMap acronym = new TreeMap();
```

```

String key;
Definition value; // acronym definition

acronym.put("CS", new Definition("Computer Science"));
acronym.put("AP", new Definition("Advanced Placement"));

Set keys = acronym.keySet();
Iterator iter = keys.iterator();
while (iter.hasNext()){
    key = (String)iter.next();
    value = (Definition)acronym.get(key);

    // process value
    System.out.println(key + " stands for " +
                        value.getDefinition());
}

```

The output for this code fragment is

```

AP stands for Advanced Placement
CS stands for Computer Science

```

The values will be processed in the ascending order of keys.

5. `TreeMap` is more general than `TreeSet`. Both implement Binary Search Trees, but in `TreeSet` the values are compared to each other, while in `TreeMap`, no ordering is assumed for the values and the tree is arranged according to the order of the keys. In a way, a set is a special case of a map where a value serves as its own key.

```

import java.util.TreeMap;

TreeMap <Integer, String> map = new TreeMap <Integer, String> ();
String name = "John";
map.put(new Integer(3), name);
map.put(new Integer(2), "Nancy");
map.put(new Integer(1), "George");

System.out.println("Size of map" + map.size());
System.out.println("Person for 1: " + map.get(1));

```

G. HashMap

page 9 of 11

1. The `java.util.HashMap` class implements the `Map` interface using a hash code into an array.

```
HashMap <Integer, String> familyMap = new HashMap <Integer, String> ();  
String name = "John";  
familyMap.put(new Integer(3), name);  
familyMap.put(new Integer(2), "Nancy");  
familyMap.put(new Integer(1), "George");  
  
Set <Integer> familyList = new TreeSet <Integer>();  
familyList = familyMap.keySet();
```

Summary/Review

page 10 of 11

The `java.util` package includes two interfaces, `Set` and `Map`. The `TreeSet` and the `TreeMap` classes implement their interfaces using balanced binary trees. If the data implements `Comparable`, the data will be stored in order. The `HashSet` and `HashMap` classes implement the `java.util.Map`.

Chapter 29

AB29 Introduction

page 1 of 16

In this lesson, you will study a new data structure, the linked list, which is used to implement a list of elements arranged in some kind of order. This is the backbone of the `LinkedList` class that is part of the Java Collections. The linked list structure uses memory that shrinks and grows as needed but not in the same way as arrays. The discussion of linked lists includes the specification and implementation of a node class. This curriculum will use the node class suggested by College Board.

This lesson will present the common manipulations performed on a linked list: adding, removing and traversing. The concepts from this lesson will provide us with the tools to build our own more advanced data structure, the binary search tree.

The key topics for this lesson are:

- A. [Declarations for a Linked List](#)
- B. [Methods for Manipulating Nodes](#)
- C. [Implementing Linked Lists](#)
- D. [Traversing a Linked List - Method printList](#)
- E. [Pitfalls of Linked Data Structures](#)
- F. [Building an Ordered Linked List](#)
- G. [Linked List Algorithms](#)
- H. [Static vs. Dynamic Data Structures](#)
- I. [Doubly-Linked Lists](#)
- J. [Deleting from a Doubly-Linked List](#)

[AB29 Vocabulary](#)

page 2 of 16

AUXILIARY POINTER

DOUBLY-LINKED LIST

EXTERNAL POINTER

INTERNAL POINTER

LINKED LIST

NODE

NULL REFERENCE

TRAVERSE

A. Declarations for a Linked List

page 3 of 16

1. A linked list is a sequence of elements arranged one after another, with each element connected to the next element by a “link.” The link to the next element is combined with each element in a component called a *node*. A node is represented pictorially as a box with an element written inside of the box and a link drawn as an arrow and used to connect one node to another.

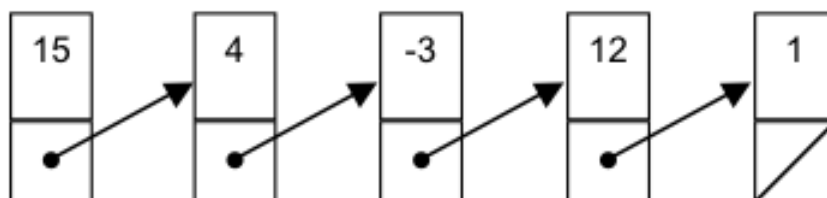


Figure 29-1

In addition to connecting two nodes, the links also place the nodes in a particular order. In Figure 29-1 above, the five nodes form a chain with the first node linked to the second, the second node linked to the third node, and so on until the last node is reached. The last node is a special case since it is not linked to another node and its link is indicated with a diagonal line.

2. Each node contains two pieces of information: an element and a reference to another node. This can be implemented as a Java class for a node using an instance variable to hold the element, and a second instance variable that is a reference to another node as follows.

```
public class ListNode{  
    private Object value; // the element stored in this node  
    private ListNode next; // reference to next node in List  
    ...  
}
```

3. The declaration seems circular and in some ways it is, but the compiler will allow such definitions. A `ListNode` will have two data members, an object and a reference to another `ListNode`. The instance variable `next` will point to the next `ListNode` in a linked list.
4. The `ListNode` class is constructed so that the elements of a list are objects (i.e., have the object data type). Since any class extends `object`, you can put any kind of object into a list, including arrays and strings.
5. Whenever a program builds and manipulates a linked list, the nodes are accessed through one or more references to nodes. Typically, a program includes a reference to the first node (`first`) and a reference to the last node (`last`).

```
ListNode first;  
ListNode last;
```

A program can create a linked list as shown below in Figure 29.2. The `first` and `last` reference variables provide access to the first and last nodes of the list.

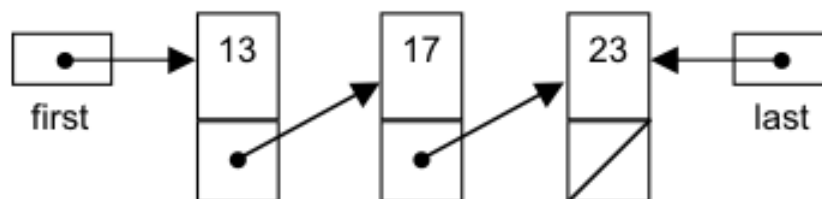


Figure 29.2

6. Figure 29-2 illustrates a linked list with a reference to the first node where the list

terminates at the final node (indicated by a diagonal line in the reference field of the last node). Instead of a reference to another node, the final node contains a *null* reference. Recall that null is a special Java constant that can be used for any reference variable that has nothing to refer to. There are several common situations where the null reference is used:

- a. When a reference variable is first declared and there is not yet an object for it to refer to, it can be given an initial value of the null reference.
- b. The null reference occurs in the link part of the final node of a linked list.
- c. When a linked list does not yet have any nodes, the null reference is used for the first and last reference variables to indicate an empty list.

B. Methods for Manipulating Nodes

page 4 of 16

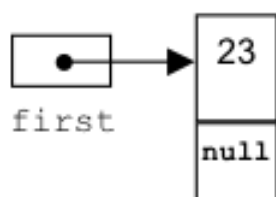
1. Methods for the `ListNode` class will consist of those for creating, accessing, and modifying nodes.
2. The constructor for the `ListNode` class is responsible for creating a node and initializing the two instance variables of a new node.

```
public ListNode(Object initValue, ListNode initNext) {  
    // post: constructs a new element with object initValue,  
    // followed by next element  
    value = initValue;  
    next = initNext;  
}
```

3. Here is an example of code to create the first node of a linked list.

```
ListNode first;  
first = new ListNode(new Integer(23), null);
```

After execution of the two statements, `first` refers to the header node of a small linked list that contains just one node with the `Integer 23`.



4. Getting and setting the data and link of the node are accomplished with getter and setter methods as follows.

```
public Object getValue(){
// post: returns value associated with this element
    return value;
}

public ListNode getNext(){
// post: returns reference to next value in list
    return next;
}

public void setValue(Object theNewValue) {
    value = theNewValue;
}

public void setNext(ListNode theNewNext) {
// post: sets reference to new next value
    next = theNewNext;
}
```

5. The following segment of code using `ListNode` will illustrate the syntax of accessing the data members of a `ListNode`.

```
ListNode list;

list = new ListNode(new Integer(13), null);

System.out.println("The node contains: " +
                    (Integer)list.getValue());

list.setValue(new Integer(17));
System.out.println("The node contains: " +
                    (Integer)list.getValue());
```

Run Output:

```
The node contains: 13
The node contains: 17
```

C. Implementing Linked Lists

page 5 of 16

1. In this section, we will look at a class that implements a linked list of `ListNode` objects.

This class encapsulates the list operations that maintain the links as the list is modified. To keep the class simple, we will implement only a singly linked list, and the list class will supply direct access only to the first list element.

2. The `SinglyLinkedList` class holds a reference, `first`, to the first `ListNode` (or null, if the list is completely empty). Access to the first node is provided by the `getFirst` method. If the list is empty, a `NoSuchElementException` is thrown (see Lesson A13, *Exceptions*).

```
public class SinglyLinkedList{
    private ListNode first;

    public SinglyLinkedList(){
        first = null;
    }

    public Object getFirst(){
        if (first == null){
            throw new NoSuchElementException();
        }else
            return first.getValue();
    }
}
```

3. Additional nodes are added to the head of the list with the `addFirst` method. When a new link is added to the list, it becomes the head of the list, and the link that the old list had becomes the next link:

```
public class SinglyLinkedList{

    ...
    public void addFirst(Object value) {
        first = new ListNode(value, first);
    }
    ...
}
```

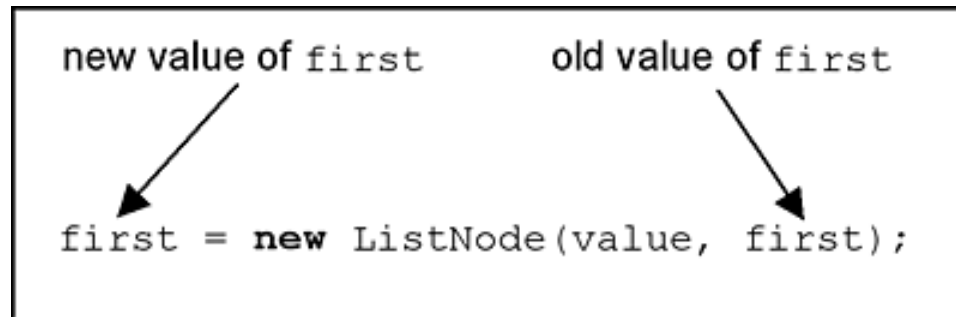
4. The statement `ListNode(value, first)` invokes the `ListNode` constructor. The line of code

```
first = new ListNode(value, first);
```

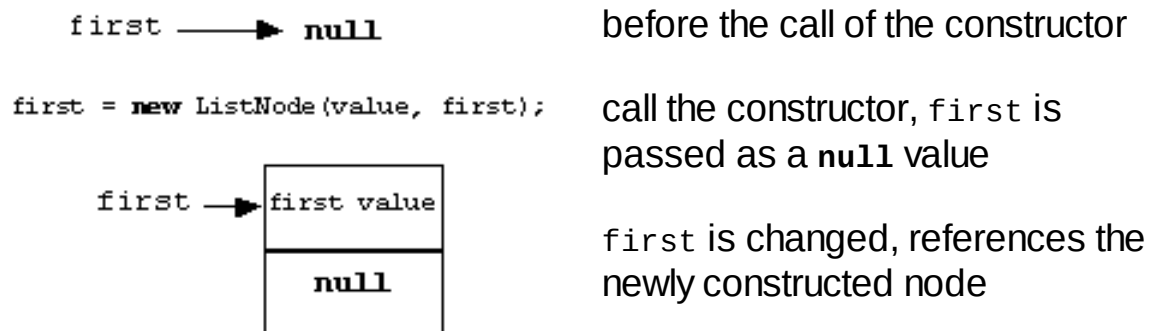
is broken down as follows.

- a. The `new` command allocates enough memory to store a `ListNode`.
- b. The new `ListNode` will be assigned the values of `value` and `first`
- c. The address of this newly constructed `ListNode` is assigned to `first`.

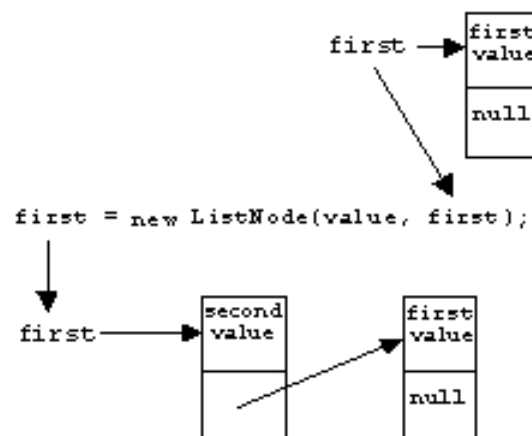
d. It is important to understand the old and new values of `first`:



5. The very first time that `addFirst()` is called, the instance variable, `first`, will be `null`. A new node is constructed with the values `value` and `null` and now `first` points to this new node. The constructor provides a new node between the variable `first` and the node that `first` formerly referenced.



6. The second time that `addFirst()` is called, `first` is already pointing to the `first` node of the linked list. When the constructor is called, the new node is constructed and placed between `first` and the node `first` used to point to.



The value of `first` passed to the `ListNode` constructor is used to initialize the next field of the new node.

1. To traverse a list means to start at one end and visit all the nodes. In the case of method `printList`, the task is to print the `value` field from each node.

```
public class SinglyLinkedList{

    ...
    public void printList(){

        ListNode temp = first; // start at the first node
        while (temp != null) {
            System.out.print(temp.getValue() + " ");
            temp = temp.getNext(); // go to next node
        }
    }
    ...
}
```

- a. We need a variable to traverse through the list so `temp` is created. Because `temp` is an alias to `first`, we can use it to traverse the list without altering the reference to the start of the list. The `ListNode` variable, `temp`, will contain `null` when we are done.
- b. Until `temp` equals `null`, the `while` loop will do two steps at each node; print the data field, then advance the `temp` reference.
- c. The statement, `temp = temp.getNext()`, is a very important one, this moves `temp` to the next node.

1. A linked list must end with a `null` value. Without such a marker at the end of the list, a routine cannot “see” the end of the data structure. This assignment of a `null` value at the end of the list is often taken care of when a new node is packaged or through the use of a constructor.
2. When a reference variable is `null`, it is a programming error to invoke one of its

methods or to try to access one of its instance variables. For example, a program may maintain a reference to the first node of a linked list, as follows:

```
ListNode first;
```

Initially, the list is empty and `first` is the null reference. At this point, it is a programming error to invoke one of the `first`'s methods. The error would occur as a `NullPointerException`.

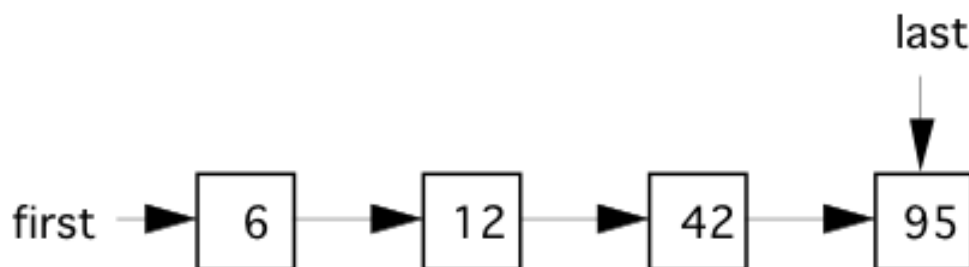
F. Building an Ordered Linked List

page 8 of 16

1. If data are supplied in unordered fashion, building an ordered linked list from the data becomes a harder problem. Consider the following cases for inserting a new value into the linked list:
 - a. Insert the new value into an empty list.
 - b. Insert the new value at the front of the list.
 - c. Insert the new value at the end of the list.
 - d. Insert the new value between two nodes.
2. Suppose the data was supplied in this order.

42 6 95 12 <eof>

The resulting ordered linked list looks like this:



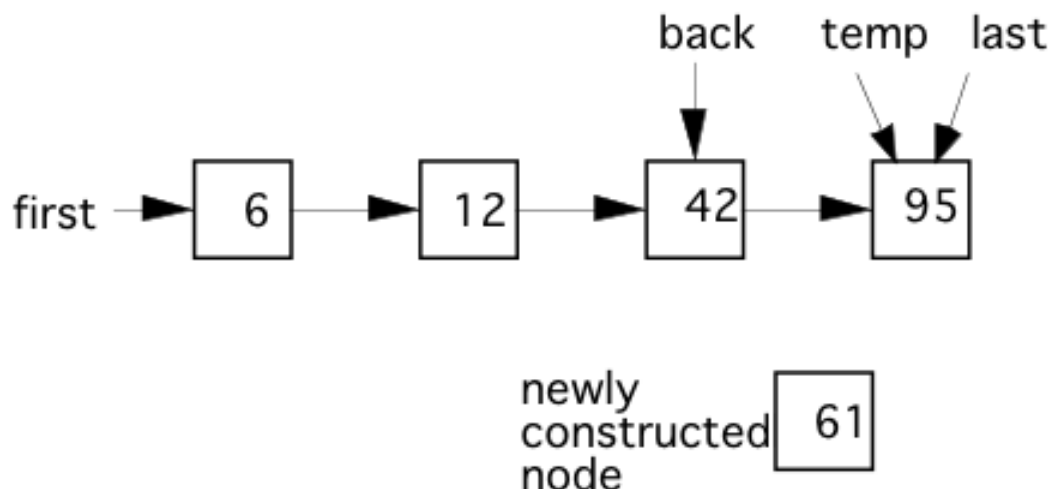
3. Two of the cases are easy to identify and solve: the empty list case (a) and placing the new value at the front of the list (b). Adding the value to the end of the list (c) is easy to

identify and solve if you maintain a marker to the last node. To assist us in our discussion of linked list algorithms, two definitions will be helpful:

- a. An internal pointer is one that exists inside a node. An internal pointer joins one node to the next in the linked list.
 - b. An external pointer is one that points to a node from outside the list. Every linked data structure must have at least one external pointer that allows access to the data structure. The linked list in this lesson has two external pointers, `first` and `last`.
4. Suppose a fifth value, 61, is to be inserted into the list (d). We can see in the diagram that it will go between the 42 and 95. When inserting a value into the list, we will use helping external pointers, also called auxiliary pointers.
5. It is possible to find the attachment point using just one auxiliary pointer, but we will use two: `temp` and `back`. Using two external pointers makes the hookup easier. Finding the attachment point involves a search. The pointer, `temp`, starts at `first` and is moved through the list until it is just past the insertion point. The pointer, `back`, trails `temp` by one position at each step of the search.

```
while we haven't found the attachment point{
    back = temp;           //move back up to temp
    temp = temp.getNext(); //advance temp one node ahead
}
```

The position of our pointers at the end of the search:



6. The new node needs to be placed between `back` and `temp`. If we call this new node `newValue`, then inserting it into the list would consist of the following steps.

```
back.setNext(newValue);
newValue.setNext(temp);
```

1. Searching an ordered linked list is a sequential search process similar to what we just covered with insertion. A linked list is not a random access data structure. You cannot jump to the middle of a linked list. Only sequential moves are possible. The search function could return a value or a pointer to that cell.
2. Deleting a value involves the following steps:
 - a. Locating the value (if it exists) to be deleted.
 - b. Re-hooking pointers around the node to be deleted.
 - c. Updating first or last if necessary.

1. An array is a somewhat static data structure that has the following advantages and disadvantages:

Advantages of an array	Disadvantages of an array
1. Easy to implement and use 2. Fast, random-access feature	1. Memory is usually wasted 2. Inserting/deleting is slower

2. A linked list (LL) is a dynamic data structure that has the following advantages and disadvantages:

Advantages of LL	Disadvantages of LL
1. Memory is allocated when the program is run; therefore, the data structure is only as big as it needs to be.	1. Each node of the list takes more memory. 2. The data structure is not random access so processing must be

I. Doubly-Linked Lists

page 11 of 16

1. The node of a doubly-linked list will contain the information field and two reference fields. One reference will refer to a previous node while the other reference will refer to the next node in the list.
2. The following class definitions will be used:

```
public class DListNode{
    private Object value;
    private DListNode next;
    private DListNode previous;

    // Constructor:
    public DListNode(Object initValue,
                    DListNode initNext,
                    DListNode initPrevious){

        value = initValue;
        next = initNext;
        previous = initPrevious;
    }

    public Object getValue(){
        return value;
    }

    public DListNode getNext(){
        return next;
    }

    public DListNode getPrevious(){
        return previous;
    }

    public void setValue(Object theNewValue){
        value = theNewValue;
    }

    public void setNext(DListNode theNewNext){
        next = theNewNext;
    }

    public void setPrevious(DListNode theNewPrevious){
```

```

        previous = theNewPrevious;
    }
}

```

- Figure 29-3 illustrates a doubly-linked list, of type `DListNode` containing `Integer` objects.

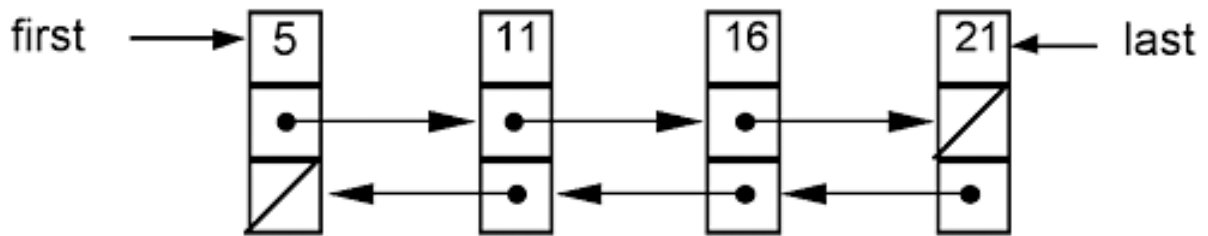


Figure 29-3

- A `null` value must be placed at each end of the list to signify the end of the data structure. In the diagram, a `null` is indicated with the diagonal line.
- A doubly-linked list should have two external references to access the data structure. In the case above, `first` and `last` are the two entry points.
- A doubly-linked list can be traversed in either direction.
- Inserting values into an ordered doubly-linked list is a similar process to the algorithm used with a singly-linked list. However, the number of reference manipulations will double.
- The addition of a new node to a position between two existing nodes will require four reference hookups.

J. Deleting from a Doubly-Linked List

page 12 of 16

- The same special cases that we saw for singly-linked lists also apply to doubly-linked lists.
- Deletions require updating both the `next` and `previous` pointers.

You have just learned your first dynamic data structure, a linked list. The concept of indirection makes dealing with references a bit more difficult, but careful reading and lots of diagrams will help. Following a working program is a good start, but only by writing code will you develop proficiency with lists. The doubly-linked list is an extension of the basic linked list.

Chapter 30

AB30 Introduction

page 1 of 15

A binary tree is a different kind of data structure that demands new terminology and algorithms. A binary tree node will have two pointers available for linking with other nodes, resulting in diagrams that look like inverted trees. A binary tree will begin with one node at the top and branch out below. As you might expect, the potential of going one of two different ways leads to some challenging programming problems. In fact, the idea of trying one direction and then backtracking naturally leads to recursion.

The key topics for this lesson are:

- A. [Binary Tree Vocabulary](#)
- B. [Building a Binary Tree](#)
- C. [Shape of a Binary Tree](#)
- D. [Inorder Tree Traversal](#)
- E. [Preorder and Postorder Tree Traversals](#)
- F. [Counting the Nodes in a Tree](#)
- G. [Searching a Binary Tree](#)
- H. [Deletion from a Binary Tree](#)
- I. [deleteTargetNode Method](#)

[AB30 Vocabulary](#)

page 2 of 15

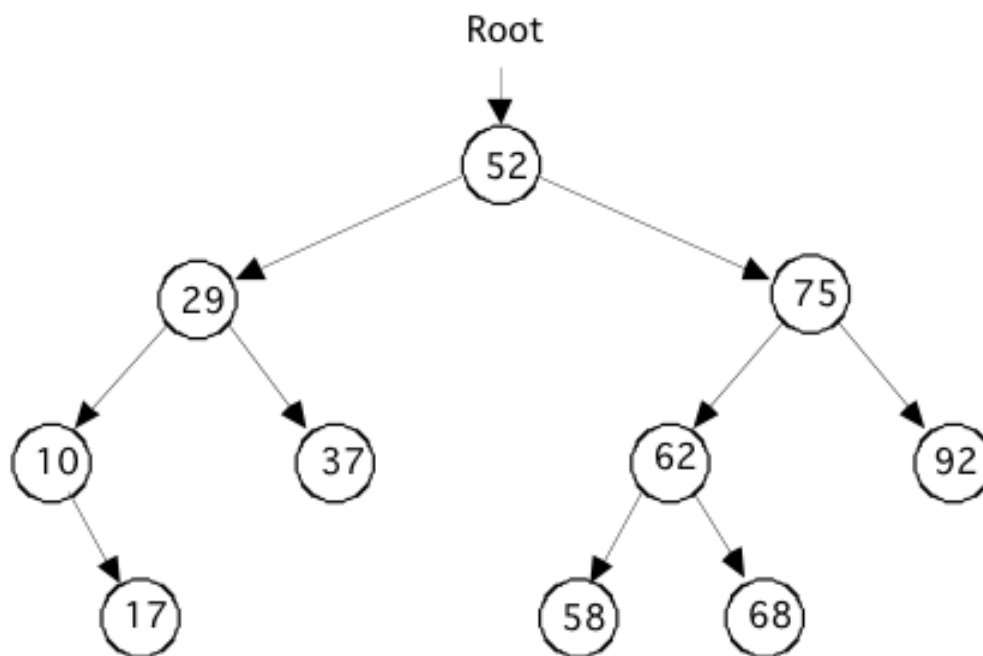
BINARY TREE
EDGE
LEAF
PARENT NODE
PREORDER
SUBTREE
VISITING A NODE

CHILD NODE
INORDER
POSTORDER
ROOT NODE
TREE TRAVERSAL

A. Binary Tree Vocabulary

page 3 of 15

1. A binary tree is a data structure where each node has two pointers, each pointing to another node or containing a **nu11** value.



2. The following binary tree terms will be defined and applied to the above example.
 - a. Root node - the top node in the tree; the node whose value is 52.
 - b. Parent node - a node that points to one or two nodes below it.
 - c. Child node - the node being pointed to by a parent; every node in the tree is a child to another node, except for the root node.

- d. Leaf - a node that has no children
 - e. Level - the distance from the root, calculated by counting the shortest distance from the root to that node. Examples: 29 is stored in a node at level 1, 62 is stored in a node at level 2, 17 is stored at level 3, etc.
 - f. Edge - an edge joins two nodes. In the above diagram, each arrow represents an edge.
- 3. This tree is an example of an ordered binary tree that has the following property. For every parent node, a child to the right will have a larger value, while a child to the left will have a smaller value.
 - 4. A subtree is the entire left branch or right branch of a node. For example, the left subtree of the node containing 52 has 4 nodes. The right subtree of node containing 75 has only 1 node.
 - 5. A leaf will have two **nu11** pointers.

B. Building a Binary Tree

page 4 of 15

- 1. The following definition of a `TreeNode` class will be used in the remainder of this section on building a binary tree.

```
public class TreeNode{
    private Object value;
    private TreeNode left;
    private TreeNode right;

    public TreeNode(Object initValue, TreeNode initLeft,
        TreeNode initRight) {
        value = initValue;
        left = initLeft;
        right = initRight;
    }

    public Object getValue(){
        return value;
    }

    public TreeNode getLeft(){
        return left;
    }
}
```

```

    }

    public TreeNode getRight(){
        return right;
    }

    public void setValue(Object theNewValue) {
        value = theNewValue;
    }

    public void setLeft(TreeNode theNewLeft) {
        left = theNewLeft;
    }

    public void setRight(TreeNode theNewRight) {
        right = theNewRight;
    }
}

```

2. Suppose the following integers were inserted into a sorted binary tree in the indicated order.

26 79 14 99 53 9 35 21 87

Draw the resulting binary tree.

3. You will notice that as each node was added to the tree, it was inserted as a leaf. The insert algorithm will be recursive.

See [Transparency AB30.1](#), *Building a Binary Tree*

4. Given this parameter list for the insert method, develop the pseudocode below it.

```

void insert (TreeNode node, Object data)
// Will insert data into an ordered binary tree.
// The solution is recursive.

```

C. Shape of a Binary Tree

page 5 of 15

1. The shape of a binary tree will affect its performance as a data structure and is dependent on the initial order of the data set used to build the tree.

2. If the data set is already sorted (1 2 3 4...), the binary tree is essentially a linked list with an unused left pointer in each node.
3. A data set in random order will lead to a more balanced tree.
4. Ideally, we want binary trees that are balanced with almost equal numbers of nodes in each subtree of every node. A balanced binary tree is defined as follows:

For every node in the tree, the number of nodes in its left subtree is equal to the number of nodes in its right subtree, plus or minus one.

Please note: Balancing binary trees will not be covered in this curriculum.

D. Inorder Tree Traversal

page 6 of 15

1. At first glance, printing out the information in a binary tree, in ascending order, does not appear to be a simple task. The example diagram in Transparency AB30.1, *Building a Binary Tree* illustrates that the first node value printed should be 9, and getting there is fairly simple. The next value is 14, then 21. Then, there's a major issue - how do we get back to the root node whose value is 26? This is a backtracking problem that is elegantly solved using recursion.
2. A tree traversal is an algorithm that visits every node in the tree. To visit a node means to process something regarding the data stored in the node. For now, visiting the node will involve printing the *value* object field.
3. An `inorder` tree traversal visits every node in a certain order. Each node is processed in the following sequence.
 - Traverse the left subtree `inorder`
 - Visit node
 - Traverse the right subtree `inorder`

Notice that actually visiting the node occurs between the two recursive calls.

4. Here is the code for the `inorder` method.

```
void inorder (TreeNode temp) {  
    if (temp != null) {  
        inorder (temp.getLeft());
```

```

        System.out.println(temp.getValue());
        inorder (temp.getRight());
    }
}

```

5. To see how the method, inorder works, study the following table:

Step	Current Node Value
inorder is passed the root node	26
inorder is passed the left subtree of 26	14
inorder is passed the left subtree of 14	9
inorder is passed the left subtree of 9	null
Output 9	9
inorder is passed the right subtree of 9	null
Output 14	14
inorder is passed the right subtree of 14	21
inorder is passed the left subtree of 21	null
Output 21	21
inorder is passed the right subtree of 21	null
Output 26	26
inorder is passed the right subtree of 26	79
Continue processing	

E. Preorder and Postorder Tree Traversals

page 7 of 15

1. A preorder tree traversal processes each node in a different order.

- Visit the node
- Process the left subtree preorder
- Process the right subtree preorder

The only difference is that we visit the root first, then go left, then right. The preorder

output of the same binary tree will be:

26 14 9 21 79 53 35 99 87

2. A postorder tree traversal has this order:

- Process the left subtree postorder
- Process the right subtree postorder
- Visit the node

The prefix “post” refers to after, hence the location of visiting the node after the recursive calls. The printout of the same tree will be as follows:

9 21 14 35 53 87 99 79 26

F. Counting the Nodes in a Tree

page 8 of 15

1. A standard binary tree algorithm is to count the number of nodes in the tree. Here is a pseudocode version.

- Count the left subtree recursively
- Count the current node as one
- Count the right subtree recursively

2. As you develop the code, consider what base case will terminate the recursion.

G. Searching a Binary Tree

page 9 of 15

1. Searching an ordered binary tree can be solved iteratively or recursively. Here is the iterative version:

```
TreeNode find(TreeNode root, Comparable valueToFind) {  
    TreeNode node = root;
```

```

while (node != null) {
    int result = valueToFind.compareTo(node.getValue());
    if (result == 0)
        return node;
    else if (result < 0)
        node = node.getLeft();
    else // if (result > 0)
        node = node.getRight();
}
return null;
}

```

2. If the value is not in the tree, the node pointer will eventually hit a **null**.
3. Notice the type of the argument, `valueToFind`, in the `find` method is designated as `Comparable`. `find`'s code calls the `compareTo` method of the `valueToFind` object to determine the ordering relationship. We declare `valueToFind` as a `Comparable`, not just an object to guarantee that it will have a `compareTo` method.
4. A recursive version is left for you to implement as part of Lab Assignment AB30.2, *BS Tree continued*.
5. The order of searching an ordered binary tree is $O(\log_2 N)$ for the best case situation. For a perfectly balanced tree, the capacity of each level is $2^{\text{level \#}}$.

Level #	Capacity of Level	Capacity of Tree
0	1	1
1	2	3
2	4	7
3	8	15
4	16	31
5	32	63
etc.		

6. So starting at the root node, a tree of 63 nodes would require a maximum of 5 left or right moves to find a value. The number of steps in searching an ordered binary tree is approximately $O(\log_2 N)$.

H. Deletion from a Binary Tree

1. Deleting a node involves two steps:

- a. Locating the node to be deleted.
 - b. Eliminating that node from the data structure.
2. After locating the node to be deleted, we must determine the nature of that node.
- a. If it is a leaf, make the parent node point to **null**.
 - b. If it has one child on the right, make the parent node point to the right child.
 - c. If it has one child on the left, make the parent node point to the left child.
 - d. If it has two children, the problem becomes much harder to solve.

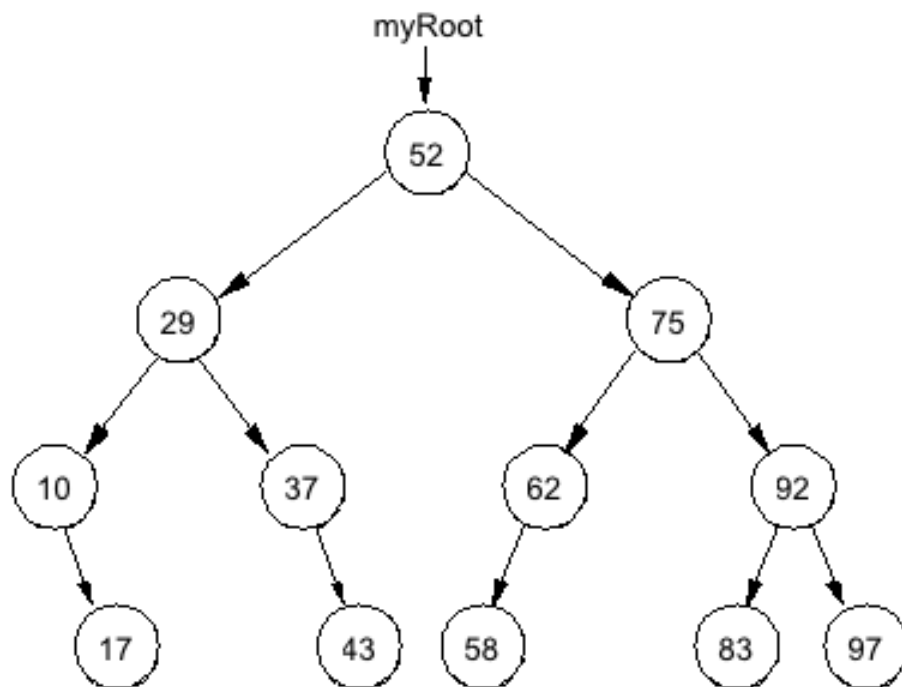


Diagram 30-1

3. The leaf node containing the value 43 will be easy to delete. The parent node of the node containing 43 will change its right pointer to **null**.
4. Deleting a node with one child on the right, like the node with value 10, will involve rerouting the node from its parent to its single right child.
5. But deleting the node with value 29, a node with two children, involves breaking off subtrees and reattaching them at the proper location.
6. The code to implement deletion from a binary tree is given in [Handout AB30.1](#), *Deletion from a Binary Tree*. The recursive `deleteHelper` method that locates the node to be deleted is given below:

```

public void delete(Comparable target){
    myRoot = deleteHelper(myRoot, target);
}

```

```

}

private TreeNode deleteHelper(TreeNode node, Comparable target){
    if (node == null) {
        throw new NoSuchElementException();
    }
    else if (target.equals(node.getValue())){
        return deleteTargetNode(node);
    }
    else if (target.compareTo(node.getValue()) < 0)
        node.setLeft(deleteHelper(node.getLeft(), target));
        return node;
    }
    else{ //target.compareTo(root.getValue()) > 0
        node.setRight(deleteHelper(node.getRight(), target));
        return node;
    }
}
}

```

7. The delete method passes the root of the tree (myRoot) and the target item to be located to the deleteHelper method. The deleteHelper method receives a TreeNode reference alias (node). The deleteHelper method has 4 scenarios:
- a. node == null, the value does not exist in the tree, throw a NoSuchElementException.
 - b. We found the correct node (target.equals(node.getValue())), call deleteTargetNode and pass it node.
 - c. Did not find the node yet, recursively call deleteHelper and pass it the internal reference to the left child.
 - d. Recursively call deleteHelper and pass it the internal reference to the right child.

I. deleteTargetNode Method

page 11 of 15

1. The deleteHelper method finds the node to be deleted and calls removeTargetNode, passing a reference to the TreeNode target as shown in the following method:

```

private TreeNode deleteTargetNode(TreeNode target){
    if (target.getRight() == null) {
        return target.getLeft();
    }
    else if (target.getLeft() == null) {
        return target.getRight();
    }
    else if (target.getLeft().getRight() == null) {

```

```

        target.setValue(target.getLeft().getValue());
        target.setLeft(target.getLeft().getLeft());
        return target;
    }
    else{ // left child has right child

        TreeNode marker = target.getLeft();
        while (marker.getRight().getRight() != null)
            marker = marker.getRight();
        target.setValue(marker.getRight().getValue());
        marker.setRight(marker.getRight().getLeft());
        return target;
    }
}

```

2. The algorithm for deletion employed in the `deleteTargetNode` method is.
 - a. Node to be deleted has no left (or right) subtree (one child). Make the link from the parent refer to the left (or right) subtree. Note that for a leaf node the link from the parent will be set to null.
 - b. Node to be deleted has non-empty left and right subtrees (two children). Change the node value to the largest value in the left subtree, and then delete the largest value from the left subtree. (The deletion of the largest value must be either scenario a or b above.)
3. The leaf and one child cases are handled in `deleteTargetNode` as follows:

```

...
if (target.getRight() == null){
    return target.getLeft();
}
else if (target.getLeft() == null){
    return target.getRight();
}
...

```

These cases are left for you and your teacher to trace.

4. The two-child case is more difficult and involves changing the node value to the largest value in the left subtree, then deleting the largest value from the left subtree. The rightmost node will be the node with the greatest value in the left subtree.
5. In Diagram 30-2 below, we will work with a smaller version of the same binary tree that we used in Diagram 30-1. Here's what happens if we wish to delete the node with value 75.

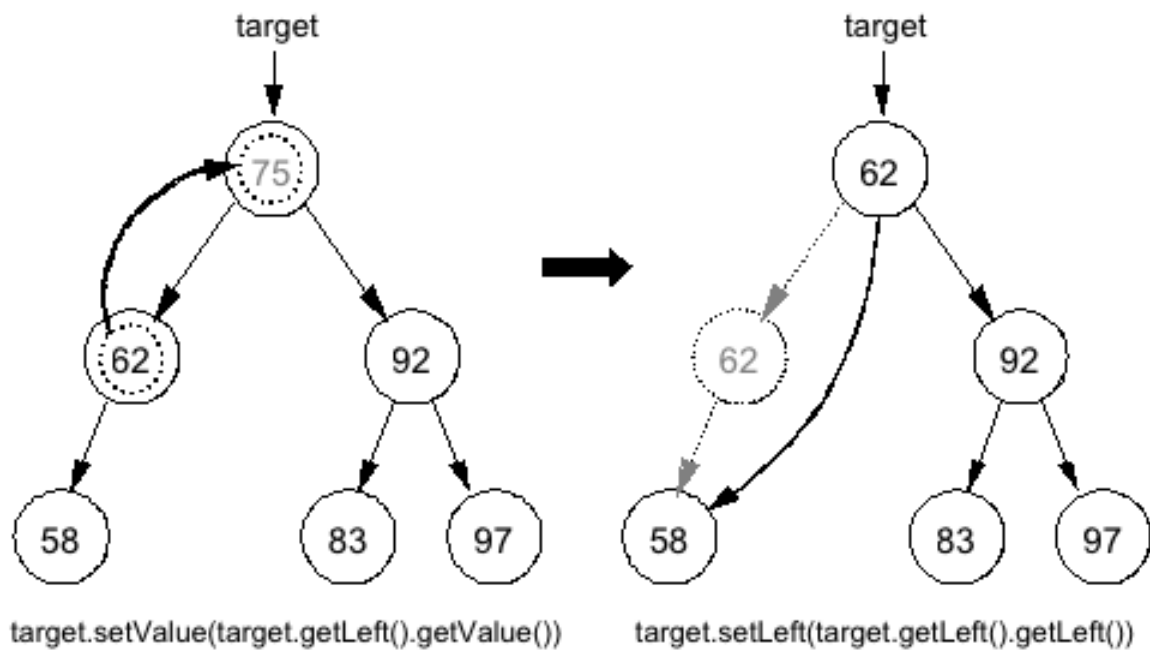


Diagram 30-2

6. Here are the steps for deleting a node having two children in which the left child has no right.

- a. Copy the contents of the left child of target and set it as the current value.

```
target.setValue(target.getLeft().getValue());
```

As shown in Diagram 30-2 above, the value 75 is replaced with 62.

- b. Reattach the left subtree to maintain an ordered tree. The left subtree of the node reference by target will now point to the node containing the value 58.

```
target.setLeft(target.getLeft().getLeft());
```

As shown in the Diagram 30-2 above, since the node that originally contained the value 62 is no longer referenced, it is removed (garbage collected).

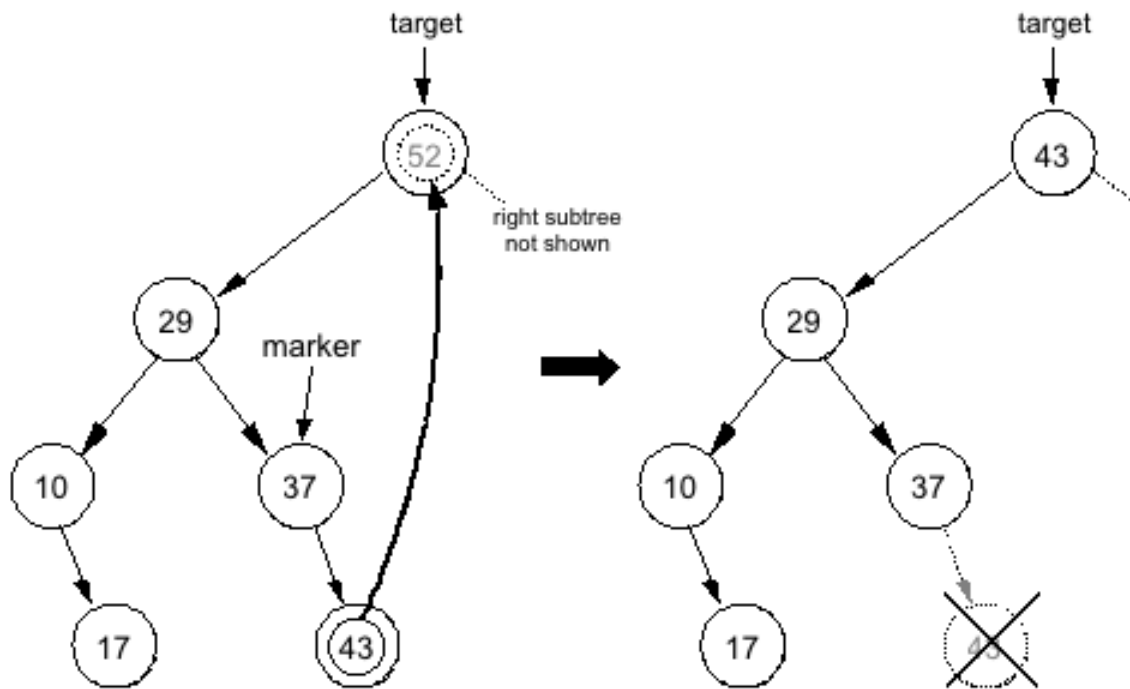


Diagram 30-3

7. In Diagram 30-3 above, we will work with the left subtree of the same binary tree that we used in Diagram 30-1. Here are the steps for deleting a node containing the value 52. In this case, the node has two children and the left child has a right child.

- a. Position marker to access the node with the largest value in the left subtree. This is the rightmost node in the left subtree.

```
TreeNode marker = target.getLeft();
while (marker.getRight().getRight() != null)
    marker = marker.getRight();
```

As shown in Diagram 30-3 above, marker now references the node pointing to the node with largest value in the left subtree (43).

- b. Copy the contents of the right child of marker and set it as the current value.

```
target.setValue(marker.getRight().getValue());
```

As shown in Diagram 30-3 above, the value 52 is replaced with 43.

- c. Delete the largest value from the right subtree. Reattach the right subtree to maintain an ordered tree.

```
marker.setRight(marker.getRight().getLeft());
```

As shown in Diagram 30-3 above, the node containing the value 43 is no longer referenced.

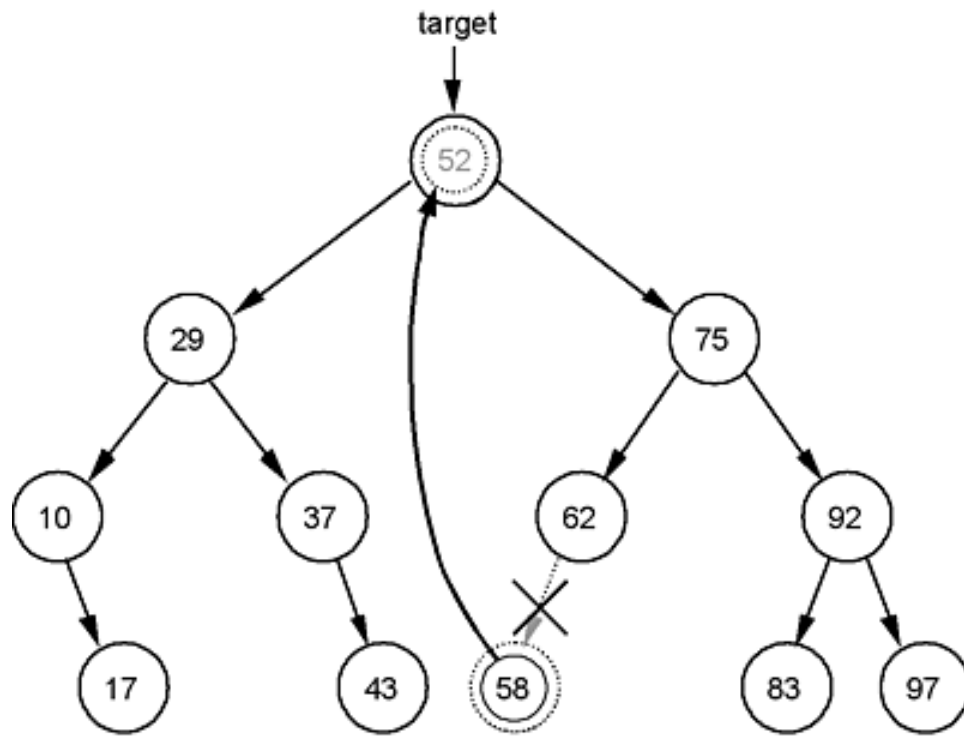


Diagram 30-4

8. This entire process for the two-child case could be directed the other way. Again, suppose the node with value 52 is to be deleted from the original tree. Referring to Diagram 30-4 above, the steps would be:
- Access the node with the smallest value in the right subtree. This is the leftmost node in the right subtree.
 - Copy the contents (58) and set it as the current value.
 - Delete the smallest value from the left subtree. Reattach the left subtree to maintain an ordered tree.

Summary/Review

page 12 of 15

This lesson introduced the Binary Search Tree and its algorithms. The most important topic of this lesson pertains to recursive algorithms used to process binary trees. An understanding of recursive tree traversals will be attained by studying the examples and drawing lots of pictures.

Chapter 31

Earlier in Lesson A9, *Recursion*, the concept of a stack was introduced. A stack is a linear data structure with well-defined insertion and deletion routines. Data is both placed and taken from the 'top' of the stack. Queues are very similar to stacks, except that Queues remove data from the front while adding data at the end.

The key topics for this lesson are:

- A. [Stacks](#)
- B. [The Java Stack Class](#)
- C. [Queues](#)
- D. [The Java Queue Interface](#)

[AB31 Vocabulary](#)

add	POP
PUSH	QUEUE
remove	STACK
TOP	

A. Stacks

1. A stack is a linear data structure.
2. All additions to and deletions from a stack occur at the top of the stack. The last item pushed onto the stack will be the first item removed. A stack is sometimes referred to as a LIFO ('Last-In, First-Out') structure.

3. Two of the more important stack operations involve pushing data onto a stack and popping data off the stack.
4. Diagram 31-1 (below) illustrates the *push* operation:

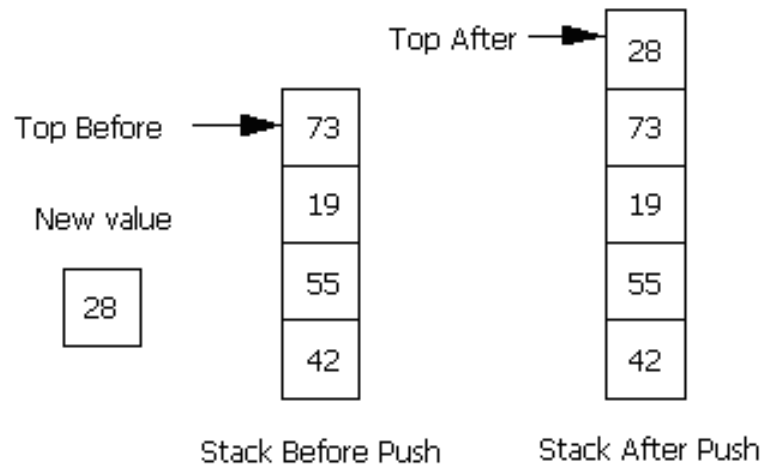


Diagram 31-1: Push Operation

5. Diagram 31-2 illustrates the *pop* operation:

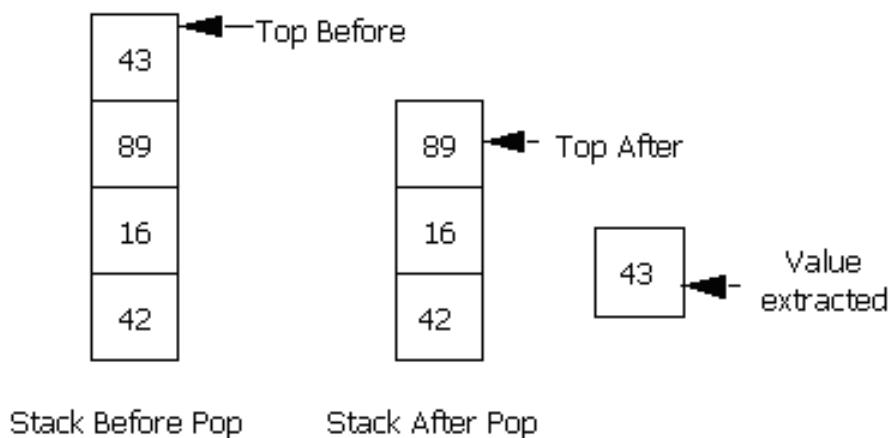


Diagram 31-2: Pop Operation

6. Stacks are very useful for implementing recursion. The local values of a method are placed on a stack each time the method calls itself. When the method returns, the stack is popped to restore the local values for that call. Another example of using a stack structure is when a web browser stores the URLs that are visited. When a new URL is visited it is placed on a stack. When the back button is used, the stack is popped.

1. The AP subset requires students to know the following methods of the `java.util.Stack`:

```
boolean empty()  
//Returns true if the Stack has no elements.  
Object peek()  
//Returns the top element without removing it.  
Object pop()  
//Returns and removes the top element.  
Object push(Object item)  
//Adds item to the top of the Stack.
```

2. To declare a reference variable for a Stack, do the following.

```
Stack <ClassName> myStack =  
    new Stack <ClassName> ();
```

3. Here is a short example showing how to create, populate, and deconstruct a Stack:

```
Stack <Integer> s = new Stack <Integer> ();  
  
for(int i = 1; i <= 5; i++){  
    s.push(i);  
}  
  
for(Integer temp : s){  
    System.out.println(temp);  
}  
  
while(!s.empty()){  
    System.out.println(s.pop());  
}
```

Output:

```
1  
2  
3  
4  
5  
5  
4  
3  
2  
1
```

Notice that the output first prints up to five and then comes back down again. The for each loop is only treating the Stack like a List, so it is not popping any of the data off the Stack. Therefore, during the while loop, the Stack is not empty and can still be

used. Also, the for each loop will start at the beginning of the List and go to the end. Because Stacks deal with all the data transactions at the top (end) of the Stack, the for each will go in the opposite order that the pop method will.

C. Queues

page 5 of 10

1. A queue is a linear data structure that is similar to waiting in line. A queue has both a front and an end.
2. Data must always enter the queue at the end and leave from the front of the line. This type of action can be summarized as FIFO ('First-In, First-Out').
3. A queue is the appropriate data structure when simulating waiting in line. A printer that is part of a multi-user network usually processes print commands on a FIFO basis. A queue would be used to maintain the order of the print jobs.

D. The Java Queue Interface

page 6 of 10

1. The AP subset requires students to know the following methods of the `java.util.Queue` interface:

```
void add(Object item)

//Adds item to the end of the Queue.
boolean isEmpty()
//Returns true if the Queue has no elements.
Object peek()
//Returns the top element without removing it.
Object remove()
//Returns and removes the first element.
```

2. To declare a reference variable for a Queue, do the following:

```
Queue <ClassName> myQ =
    new LinkedList <ClassName> ();
```

3. This is an easy way to create a new Queue object without having to create a whole

new class to implement the Queue interface. There are other ways to implement a Queue but this is the only way that AP requires.

4. Here is a short example showing how to create, populate, and deconstruct a Queue.

```
Queue <Integer> q = new LinkedList <Integer> ();

for(int i = 1; i <= 5; i++){
    q.add(i);
}

for(Integer temp : q){
    System.out.println(temp);
}

while(!q.isEmpty()){
    System.out.println(q.remove());
}
```

Output:

```
1
2
3
4
5
1
2
3
4
5
```

In this example, both loops print the exact same thing. This is because a Queue adds data to the end and takes data from the front, so it is going from the front to the end when removing data, just like the for each loop.

The stack is what makes recursive algorithms possible. In Lab Assignment AB31.1, *Inorder*, a better understanding of the recursive *inorder* function used to traverse a binary tree will be gained.

Chapter 32

In the labs in previous lessons, you have searched a data file containing ID and inventory information in a variety of ways. Of the search algorithms studied, the fastest search algorithm was $O(\log_2 N)$ for a binary tree or binary search of an ordered array. It is possible to improve on these algorithms, reducing the order of searching to $O(1)$. The data structure used to accomplish this is called a hash table, and the $O(1)$ search is referred to as hashing.

An example of hashing frequently occurs in schools, when students line up into 26 lines, each line representing the first letter of their last name. Another example occurs when one large group or list of student schedules is broken down into smaller groupings or lists, often by grade level, for easier distribution. This is what hash-coded data storage is about - breaking up and reorganizing one big list into many smaller lists.

The key topics for this lesson are:

- A. [Hashing Schemes](#)
- B. [Dealing with Collisions](#)
- C. [Order of a Hash-Coded Data Search](#)
- D. [HashSet and HashMap](#)

[AB32 Vocabulary](#)

COLLISIONS

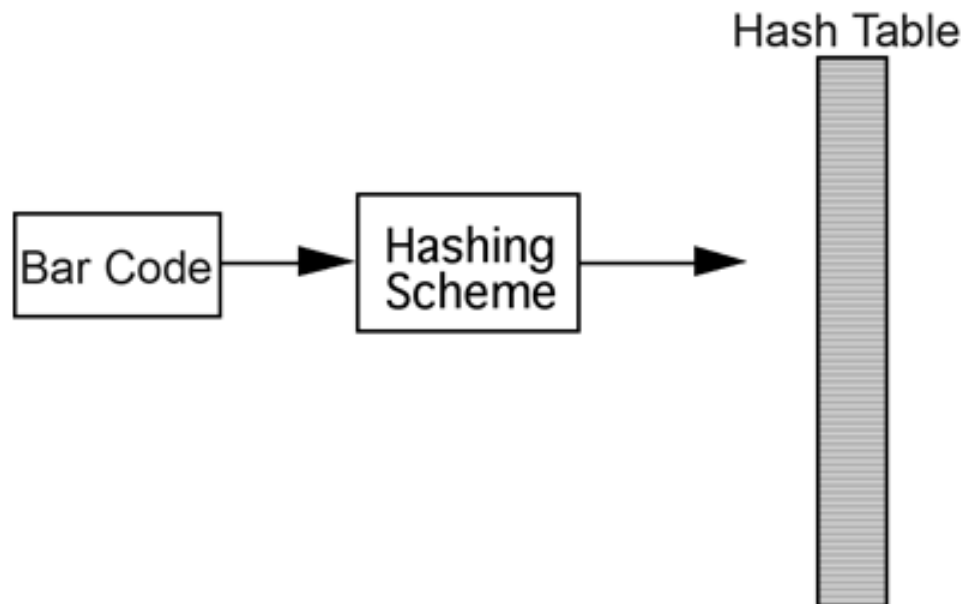
HASH KEY

HASHING

HASH TABLE

1. The example of distributing student schedules illustrates a natural means of hashing. The process can be simplified even further by organizing the schedules into piles by the first letter of the last name.
2. A hashing scheme involves converting a key piece of information into a specific location, thus reducing the amount of searching in the data structure. Instead of working with the entire list or tree of data, the hashing scheme tells you exactly where to store that data or search for it.
3. One important bar code system used by retail stores is the UPC A code¹, which involves a sequence of 10 digits. This system provides for 10 billion different possible products, 0 - 9,999,999,999 (which equals $10^{10} - 1$). For quick access, an array of 10 billion locations would be nice, but wasteful in terms of computer memory. Since it is unlikely that a store would carry such a huge number of items, a system is needed to store a list of products in a reasonably sized array.
4. A cash register using a bar code scanner needs a very quick response time when an item is scanned. The 10-digit bar code is read, the item is searched for in the store's database, and the price is returned to that register. While searching algorithms of the order ($\log_2 N$) are relatively fast, we may want an even faster algorithm.
5. Suppose hypothetically that a store maintains a database of 10,000 bar codes out of the possible 10 billion different values. The values are stored in an array called a hash table. Because an array has direct random access to every cell, using a hashing scheme will give much faster access to the desired item. The hash table is usually sized about 1.5 to 2.0 times as big as the maximum number of values stored. (The reason for this sizing will be readily apparent.) Therefore, the store will need an array with about 15,000 locations.
6. The hashing scheme tells us where item XXXXX XXXXX is stored. A hashing algorithm is a sequence of steps, usually mathematical in nature that converts a key field of information into a location in the hash table.





7. These “key-to-address transformations” are called hashing functions or hashing algorithms. When the key is composed of character data, a numerical equivalent such as the ASCII code is used so that the mathematical processing can take place. Bar codes are numbers, so conversion is not necessary. Some common hash functions:
- Division. The key is subject to integer modulus (often a prime) equal to or slightly smaller than the desired size of the array. The result of the division determines which short list to work with in the hash table.
 - Midsquare. The key is squared and the digits in the middle are retained for the address. This probably would not work well with bar codes because they are such large numbers.
 - Folding. The key is divided into several parts, each of which is combined and processed to give an address. For example:

If the bar code = 70662 11001

1) group into pairs: 70 66 21 10 01

2) multiply the first three numbers together:

$$70 \times 66 \times 21 = 97020$$

3) add this number to the last two numbers:

$$97020 + 10 + 01 = 97031$$

4) find the remainder of modulo division by 14983 (the largest prime less than

15000):

$97031 \% 14983 = 7133$

5) address 7133 is the location to store bar code 70662 11001

6) In the address 7133 will be stored all the fields related to this item, such as price and name of the item.

8. It is important to develop a good hashing function that avoids collisions in the hash table. Even when using a prime number for the divisor, it is possible for two bar codes to result in the same address, e.g. 7133. To reduce the chances of such a “collision,” the hash table is sized about 1.5-2.0 times the number of expected items. If the hash table in our example were sized at 10,000 (the number of items in the database) the likelihood of collisions would be increased. Try to balance the need for decreasing the number of collisions against memory limitations, hence the recommended sizing.
9. This advance sizing of the hash table affects the mathematics of the hashing algorithm; therefore the programmer must have a very clear idea of the number of items to be stored. The number of items must be known in advance and this number must be fairly constant during the life of the program. This limits the use of hashing to certain situations. If the number of items is unknown or varies greatly, hashing is inappropriate.

¹ Samples of bar code graphics and an explanation of how to read bar codes may be found at <http://www.adams1.com/pub/russadam/upccode.html>

B. Dealing with Collisions

page 4 of 8

1. There are two methods of creating multiple storage locations for the same address in the hash table. One solution involves a matrix, while the other uses dynamic linked lists.
2. To implement the hash table as a matrix, an estimate of the maximum number of collisions at any one address must be made. The second dimension is sized accordingly. Suppose that number is estimated as 5:

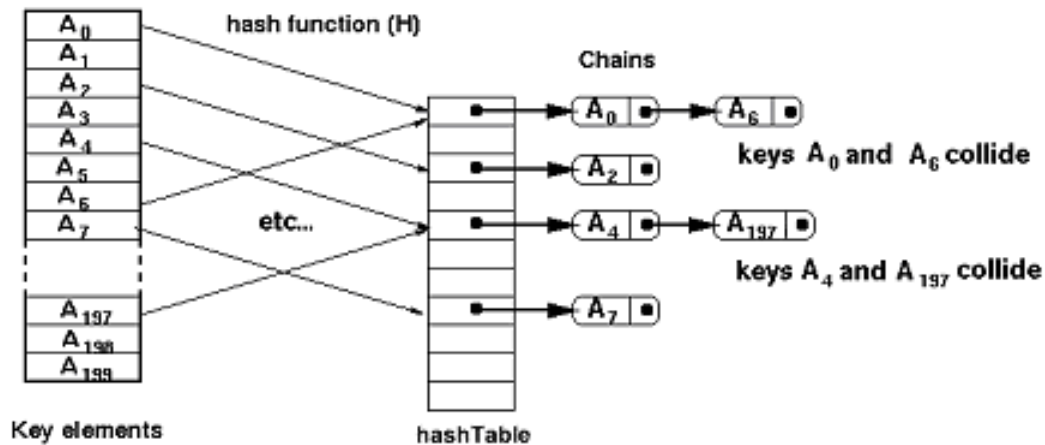
```
Item[][] table = new Item[15000][5];
```

3. This method has some major drawbacks. The size of this data structure has suddenly

increased by a factor of five. The above table will have 75,000 cells, many of which will be empty. Also, what happens if a location must deal with more than 5 collisions?

4. A dynamic solution, referred to as chaining, is much better. The linked lists will only grow when values are stored in that location in the hash table. In this version, the hash table will be an array of object references.

```
ListNode[] hashTable = new ListNode[MAX];
```



5. The order of the values in the linked lists is unimportant. If the hashing scheme is a good one, the number of collisions will be minimal.

C. Order of a Hash-Coded Data Search

page 5 of 8

1. After scanning an item at a cash register, the number of steps required to find the price is constant:
 - a. Hash the bar code value and get the hash table location.
 - b. Go to that location in the hash table, and traverse the linked list until the item is found.
 - c. Return the price.
2. The number of steps in this algorithm is constant. The hashing scheme tells the program exactly where to look in the hash table; therefore, this type of search is independent of the amount of data stored. We categorize this type of algorithm as constant time, or $O(1)$.

3. If the linked lists get lengthy, this could add a few undesirable extra steps to the process. A good hashing scheme will minimize the length of the longest list.
4. An interesting alternative to linked lists is the use of ordered binary trees to deal with collisions. For example, the hash table could consist of 10,000 potential binary trees, each ordered by a key field.
5. Remember that determining the order of an algorithm is only a categorization, not an exact calculation of the number of steps. A hashing scheme will always take more than one step, but the number of steps is independent of the size of the data set, hence it is called $O(1)$.

D. HashSet and HashMap

page 6 of 8

1. The `HashSet` and `HashMap` classes are implementations of the `Set` and `Map` interfaces from the Java standard class Library. A hash table is used to store their elements.
2. The methods from the `HashSet` and `HashMap` that are included in the AP Subset are shown below:

Methods of the HashSet Class included in the AP Subset*

```
// Adds the specified element to this set if it is not
// already present. Returns true if the set did not already
// contain the specified element.
boolean add(Object obj);
```

```
// Returns true if this set contains the specified element.
boolean contains(Object obj);
```

```
// Returns an iterator over the elements in this set.
// The elements are returned in no particular order.
Iterator iterator()
```

```
// Removes the specified element from this set if it is
// present. Returns true if the set contained the specified
// element.
boolean remove(Object obj);
```

```
// Returns the number of elements in this set.
int size();
```

Methods of the HashMap Class included in the AP Subset*

```
// Returns true if this map contains a mapping for the  
// specified key.
```

```
boolean containsKey(Object key);
```

```
// Returns the value to which the specified key is mapped,  
// or null if the map contains no mapping for this key.
```

```
boolean get(Object key);
```

```
// Returns a set containing the keys in this map.
```

```
Set keySet()
```

```
// Adds the key-value pair to this map. Returns the previous  
// value associated with specified key, or null if there was  
// no mapping for key.
```

```
boolean put(Object key, Object value);
```

```
// Returns the number of key-value pairs in this map.
```

```
int size();
```

3. In the `HashSet` class, a hash of the element is used to find its location in the hashtable. In the `HashMap` class, a hash of the key is used. The `hashCode` method, which exists on all objects, calculates the hash code.
4. The basic operations on `HashSet` and `HashMap` objects run in constant, $O(1)$, time due to the hash table implementation employed by the class.
5. The iterator that is returned by the `iterator` method of `HashSet` does not order the objects returned.

Summary/Review

page 7 of 8

Hashing is a great strategy for storing and searching information, especially where speed is a priority. In the hashing approach, the key is converted by some hashing function into an integer that is used as an index into a hash table. Different keys may be hashed into the same index, causing collisions. The performance and space requirements for a hash table vary depending on the implementation and collision resolution method. In the best case, a hash table provides $O(1)$ access to data, but the performance deteriorates with a lot of collisions. In the lab assignment for this lesson, students will implement a hash coded data storage scheme and determine its efficiency.

Chapter 33

A priority queue is essentially a list of items, each associated with a priority. In general, different items may have different priorities and we speak of one item having a higher priority than another. Given such a list, we can determine which is the highest (or the lowest) priority item in the list. Items are inserted into a priority queue in any arbitrary order. However, items are withdrawn from a priority queue in order of their priorities starting with the highest priority item first.

The key topics for this lesson are:

- A. [Priority Queues](#)
- B. [Heaps](#)
- C. [Heap Deletion and Insertion](#)
- D. [Storage of Complete Trees](#)
- E. [The PriorityQueue Class](#)

[AB33 Vocabulary](#)

COMPLETE TREE
PRIORITY QUEUE

HEAP
HEAPSORT

1. Often the items added to a queue have a priority associated with them: this priority determines the order in which they exit the queue - highest priority items are removed first. In this curriculum, the convention that will be followed is that the smallest value has the highest priority.
2. For example, consider the software that manages a printer. In general, it is possible for users to submit documents for printing much more quickly than it is possible to print them. A simple solution is to place the documents in a FIFO ('first in, first out') queue. In a sense, this is fair, because the documents are printed on a first-come, first-served basis.

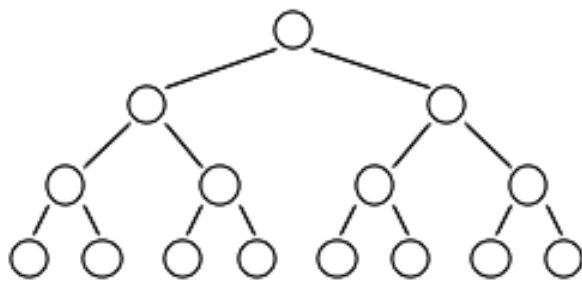
However, a user who has submitted a short document for printing will experience a long delay when much longer documents are already in the queue. An alternative solution is to use a priority queue in which the shorter a document, the higher its priority. By printing the shortest documents first, the level of frustration experienced by the users is reduced. In fact, it can be shown that printing documents in order of their length minimizes the average time a user waits for a document.

3. We can use a tree structure to keep track of the items in a priority queue - which generally provides $O(\log n)$ performance for both insertion and deletion. Unfortunately, if the tree becomes unbalanced, performance will degrade to $O(n)$ in worst cases. This will probably not be acceptable when dealing with dangerous industrial processes, nuclear reactors, flight control systems or other life-critical systems.
4. There is a structure that will provide guaranteed $O(\log n)$ performance for both insertion and deletion: it's called a *heap*.

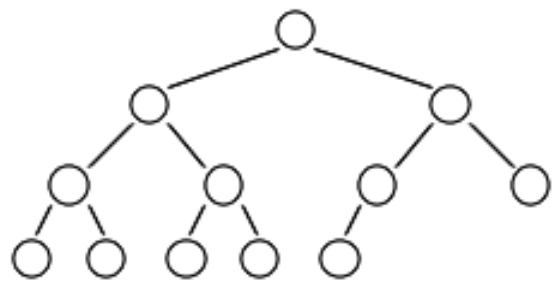
B. Heaps

page 4 of 9

1. Heaps are based on the notion of a *complete tree*. A binary tree is called *completely full* if all its levels are filled with nodes. A binary tree is completely full if it is of height h , and has $2^h - 1$ nodes. Each level contains twice as many nodes as the preceding level.
2. A binary tree is termed *complete* if it has no gaps on any level. The last level may have some leaves missing on the right, as shown below:



Full tree



Complete tree

3. A heap is a binary tree that satisfies two conditions:
 - a. it is a complete tree
 - b. the value in each node does not exceed any value in that node's left and right subtrees

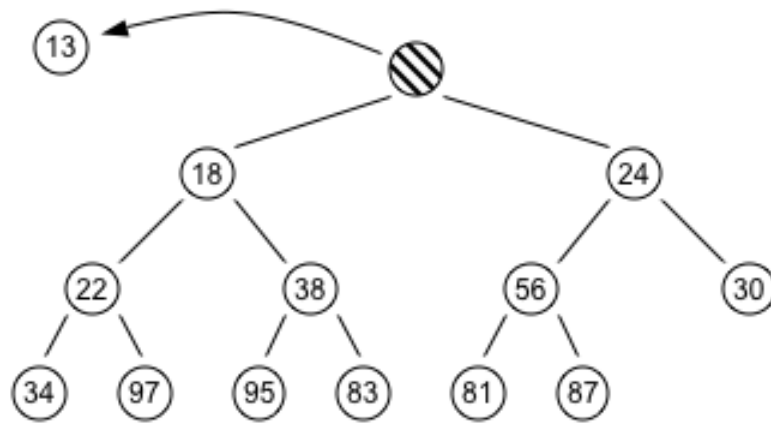
Heaps are allowed to have more than one data item with the same value, and values in the left subtree do not have to be ranked lower than values in the right subtree.

4. A heap can be used as a priority queue: the highest priority item is at the root and is trivially found. But if the root is deleted, we are left with two sub-trees and we must efficiently re-create a single tree with the heap property. The value of the heap structure is that we can both extract the highest priority item and insert a new item in $O(\log n)$ time.

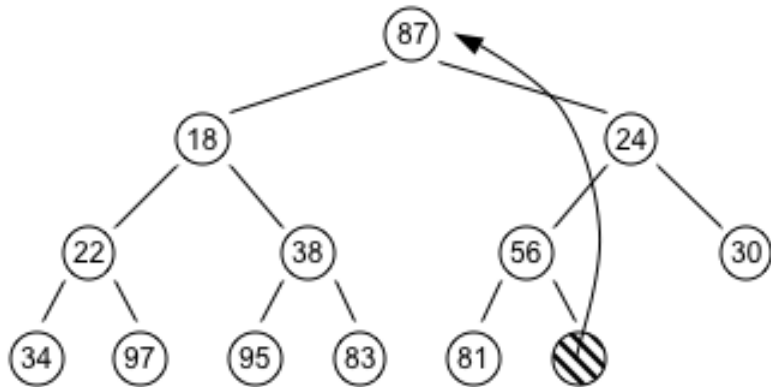
C. Heap Deletion and Insertion

page 5 of 9

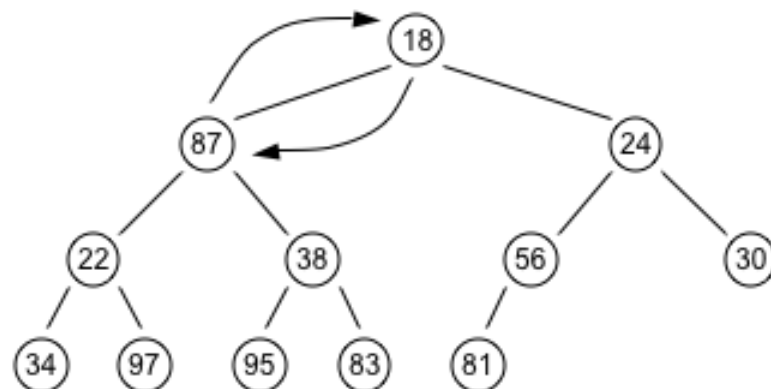
1. Removing an item from a priority queue is straightforward if the queue is represented by a binary heap. The next item to leave the queue will always be the item at the top (root) of the heap.



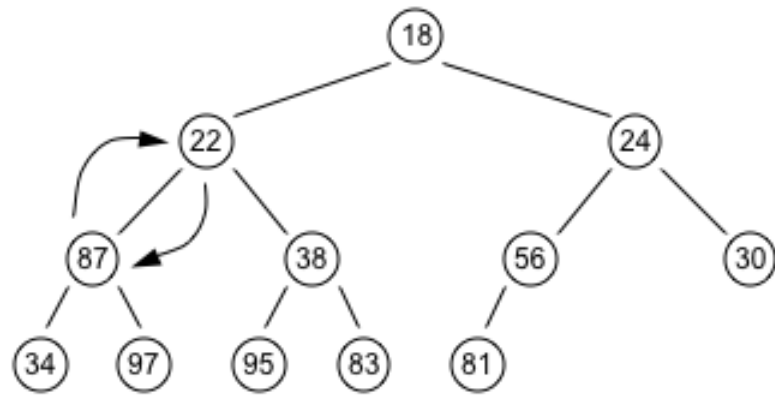
2. The shape of the heap is restored by removing the last leaf and placing it into the root. For the heap shown below, the position that must become empty is the one occupied by the 87. This is placed in the vacant root position.



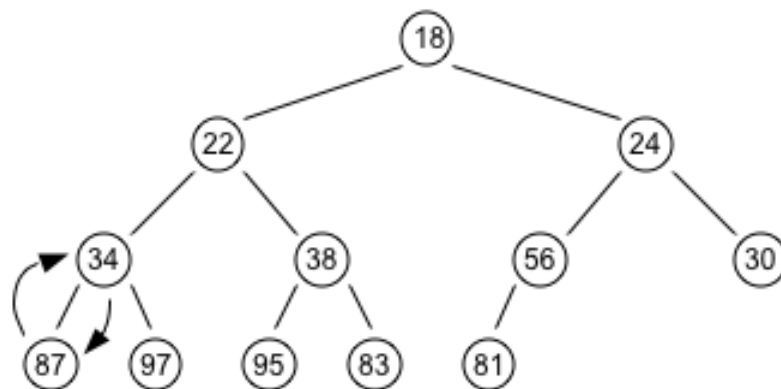
3. This has violated the condition that the root must be greater than each of its children. To repair the order, we apply the “heapify” procedure in which the value from the root moves down the heap until it falls into place.



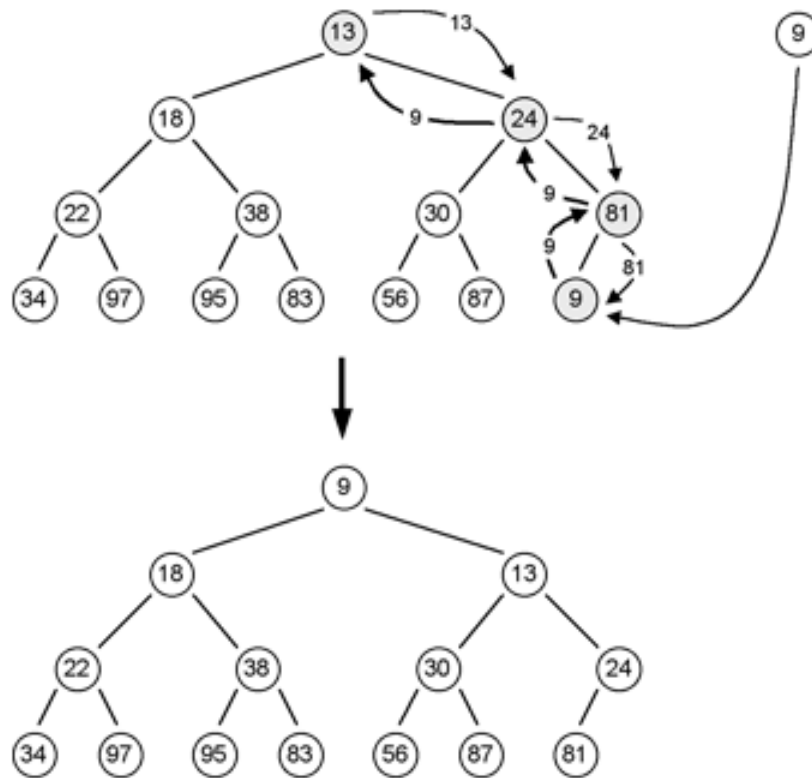
4. At each step down the value 87 is swapped with its smaller child.



5. The heap property still has not been restored in the left subtree. So again interchange the 87 with the smaller of its children.



6. We need to make at most h (recall that h is the height of the tree) interchanges of a root of a subtree with one of its children to fully restore the heap property. Thus deletion from a heap is $O(\log n)$.
7. To add an item to a heap, we follow the reverse procedure. First we add the new node as the last leaf, and then apply a “reheap up” procedure to restore the ordering property of the heap. “Reheap up” moves the new node up the tree, swapping places with its parent until the order is restored. For example, adding the value 9 to the original heap would result in the following sequence of steps:

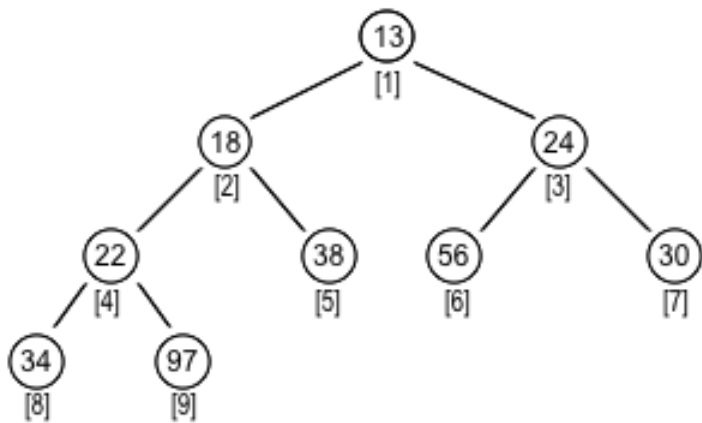


8. Again, we require $O(\log n)$ exchanges.

D. Storage of Complete Trees

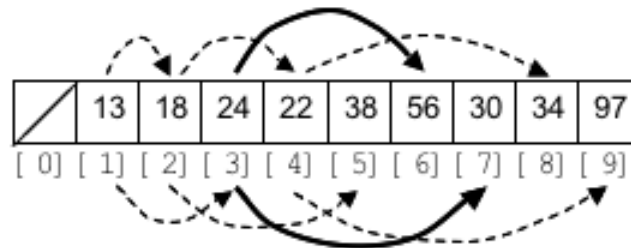
page 6 of 9

1. The properties of a complete tree lead to a very efficient storage mechanism using n sequential locations in an array.
2. An important benefit of the array representation is that we can move around the tree, from parent to children or from child to parent, by simple arithmetic. If we number the nodes from 1 at the root then
 - a. the left and right children of node i , if they are present, are at $2i$ and $2i+1$
 - b. the parent of node i is at $i/2$ (truncated to an integer)
3. If `items` is the array, the root corresponds to `Items[1]`; subsequent slots in the array store the nodes in each consecutive level from left to right.



Items[0]	<null>
Items[1]	13
Items[2]	18
Items[3]	24
Items[4]	22
Items[5]	38
Items[6]	56
Items[7]	30
Items[8]	34
Items[9]	97

4. In a Java implementation, it is convenient to leave `Items[0]` unused. With this numbering of nodes, the children of the node `Items[i]` can be found in `Items[2*i]` and `Items[2*i+1]`, and the parent of `Items[i]` is in `Items[i/2]`.



E. The PriorityQueue Class

page 7 of 9

1. The AP subset requires students to know the following methods of the `java.util.PriorityQueue`:

```

void add(Object item)
//Inserts item into the PriorityQueue.
boolean isEmpty()
//Returns true if the PriorityQueue has no //elements.
Object peek()
//Returns the next element without removing it.
Object remove()
//Returns and removes the next element according
//to its given priority. The priority is
//determined by the compareTo() method. Therefore
//all elements added to the PriorityQueue must
//implement the Comparable() interface. The
//element with the lowest value in compareTo()
//will be returned.

```

2. To declare a reference variable for a `PriorityQueue`, do the following.

```
PriorityQueue <ClassName> myStack =  
    new PriorityQueue<ClassName> ();
```

3. Here is a short example showing how to create, populate, and deconstruct a `PriorityQueue`.

```
PriorityQueue <Integer> q = new PriorityQueue <Integer> ();  
  
for(int i = 5; i >= 1; i--){  
    q.add(i);  
}  
  
for(Integer temp : q){  
    System.out.println(temp);  
}  
  
while(!q.isEmpty()){  
    System.out.println(q.remove());  
}
```

Output:

```
1  
2  
3  
4  
5  
1  
2  
3  
4  
5
```

In this example, the output prints from one to five twice. This is because the `PriorityQueue` will insert the elements it receives in the order of their priority rather than the order that they are received. Note: The `Integer` class implements `Comparable`.

This now concludes our coverage of different methods of data storage in the curriculum guide. As you continue in computer science, you will no doubt learn about other data structures and algorithms. Keep reading and learning!