# From C to Java

Some basic information to help you make the switch

by

Dr. Mark C. Lewis

## 1. Preface

One of the standard problems that we have run into since changing our curriculum so we teach C in the first semester and Java in the second is in the choice of textbooks. If we pick a CS1 text for the course it does a good job of introducing Java, but is too basic in general and doesn't cover the data structures or other advanced concepts that we want to get into. This is because it assumes you know nothing about programming, but in reality you already know how to structure the logic of a program. Instead we can pick a CS2 text and it covers all of the concepts that we will want to discuss, but such texts have virtually no information on the basics of Java programming. They all assume that you have already been programming in the language for a semester.

This pamphlet is intended to give you the basic information on Java that you need to get up to speed while growing on the knowledge you already have of C. You should read this as allotted in the schedule to help you understand the material when it is presented and the code that we will write. You should also keep it around to use as a reference through the rest of the semester.

## 2. Introduction to Object-Orientation

This course focuses on object-oriented programming and design. We just happen to teach in it Java because of the numerous benefits of that language. Up front you should understand what object-orientation is and how we use it to produce better programs.

Object-oriented programming languages have their roots in SIMULA67, a programming language that was developed to do simulations of physical systems. The

idea in SIMULA was that physical objects are more than just data, they also "encapsulate" what you can do with them. This idea of grouping data together with the functions that are allowed to work on that data is what is known as encapsulation and it is the primary aspect that joins all object oriented languages. Any entity in a program that has this binding of data and functions is typically called an object and these objects are the fundamental building blocks of object-oriented programs.

Perhaps the most pure object-oriented language is Smalltalk. Smalltalk takes the idea that you should work with objects to the absolute extreme. In Smalltalk, everything is an object, including simple numbers or boolean values. In Smalltalk, when you call a function that is associated with an object, you are said to be sending a message to the object and everything that you do in Smalltalk is done through the sending of messages, including operations as simple as adding two numbers.

Java is not as purely object-oriented as Smalltalk for a number of reasons, and when people talk about Java and C++ they typically use terms that are a bit different from that used with Smalltalk. The main way in which Java is not purely object-oriented is that primitives in Java are not objects. That is to say that Java has types for numbers, characters, and booleans that are not objects, they have no functions directly associated with them. The primitive types in Java are used much like the primitive types that you worked with in C, though there are a few differences that will be discussed in the next section. The reason for doing this was speed. Primitives are simple and typically their operations are build straight into the hardware of the machine and it is much more efficient to not add the extra overhead of making those types objects. The functions that are grouped with objects in Java are typically called **methods** and instead of saying that we pass a message to an object, we use the terminology of calling a method. The data elements that are stored in an object in Java are typically referred to as **members** or **member data**.

Java, as well as SIMULA, Smalltalk, C++, and most other object-oriented languages that you are likely to interact with, is a **class based** object oriented language. This means that objects are constructed from **classes**. One way of thinking about this is that object-oriented programs are like factories that construct various objects and have them interact as needed before throwing them away. In a class based object-oriented

language, the structure of an object is specified by its class. In the factory analogy, the class is like a blueprint, it tells the factory what parts go inside the object and how they work. The class itself does not do anything normally, it is just a layout for the objects that will do things. When we code in Java, we actually write classes. We lay out the blue prints for objects and describe how those objects interact. Some of that description can include statements that produce new objects for us to use. Once we have the objects, we can call methods on them and pass them as arguments to other methods.

The grouping of functions and data together into objects doesn't sound all that significant and hardly worth a whole class on the topic. However, it is actually sufficient to teach multiple classes on because of some of the extra abilities that come naturally when data and functions are grouped together. In fact, these other abilities are so commonly used that their use is largely folded into the term encapsulation. Encapsulation takes on new abilities and greatly enhances what we can do in programming when we provide different levels of visibility to the members of a class and the objects created from that class and when we provide a mechanism for subtyping of classes. We will talk about these topics in great depth this semester and in reasonable depth later in this document. For now I would like to present to you a high level description of what these things mean and what they mean to our programming.

The idea of allowing the programmer to set visibility on the members and methods of a class is quite simple. At the simplest level, you get to tell what things can be seen and used outside of the class and what things can only be seen or used inside of the class. A more detailed description will be given later that is specific to Java. The advantage of doing this is that code outside of a class can only depend on the things that they can see. This is referred to as **information and implementation hiding** and it leads to another valuable ability called **separation of interface and implementation**. We will see many times over the course of the semester how important these things are. When code does a good job of information hiding and separating the interface from the implementation, you can change how a class is implemented, for example, how it stores data or what data structure it is based on, without breaking the code that uses it.

Closely related to these aspects of visibility is the subtyping ability provided in Java and many other class based object oriented languages. The idea of a subtype is that

you can make one class a subtype of another so that any object of the second class qualifies as as an object of the first class as well. This provides us with an ability called **inclusion polymorphism**, which is used extensively in this course and in Java programming in general.

## 3. Basic Java Examples

One thing that you will find in working with Java is that it looks very much like C. In fact, most of the functions you wrote in C in PAD1 could be very easily converted to run in Java. Many might not need any conversion at all. However, that does not mean that your C programs would be good Java programs. The switch from C to Java is much more of a switch in paradigms and semantics instead of syntax. Part of the reason for this document is to help insure that you do not get hung up too much on the syntax during class and can focus instead on the larger ideas that are presented.

Below is a sample of Java code that prints out the numbers 0 through 9 and tells you whether each number is even or odd. In many ways, this code should look familiar to you, and after learning C you should certainly be able to read this code and understand it without any difficulties. Some of the details of what some things mean could certainly be unclear.

```
/**
 * Code Example 3.1
 * This is a simple class that has a main method in it.
 *
 * @author Mark Lewis
 */
public class MyNumbers {
	/**
	 * This is the main method for the MyNumbers class.
	 * @param args This is the array of command line arguments.
	 */
	public static void main(String[] args) {
		// This loop does the counting.
		for(int i=0; i<10; ++i) {
			System.out.print(i+" is ");
			if(i%2==0) {
				System.out.println("even.");
			} else {
				System.out.println("odd.");
			}
		}
	}
}
```

A number of things should jump out about this code. First are the similarities

with C. You see that we are declaring a function main that returns void and takes a single argument that is an array of type String and while a String in Java is not the same as a char* in C, the differences in this example aren't significant. You notice that the for loop looks almost the same as it would in C, as does the if statement. The expressions also look very similar. Things like ++i (which is roughly the same as i++, especially in this example) and i%2==0 are identical to their usage in C. One difference with the for loop is that we can declare the loop variable in the loop statement. This localizes the scope of that variable to exist only in the loop and not only prevents bugs, but can also be nice when coding in programs.

The way that we print in Java is also different than in C, but since that is part of the libraries, you expect it. This code prints using System.out.print and System.out.println. In C the dot notation was used to reference elements of structures. The dot has generally the same meaning in Java, but it is used more broadly. In Java, a dot is used to refer to something in an object, class, or package. In this case, System is a class that has a static object called out in it and this object has methods print and println that are being called.

I said that out was a static member of the class System. We also see the word static in front of the main method in our class in this example. What does that mean? Member data and functions that are modified with the keyword static are associated with the class itself, they are class variables or functions, they are not associated with the objects created for that class. Going back to the factory and blueprint analogy, a static variable is like something written on the blueprint itself. It doesn't go into every object that is built from that blueprint. Static methods can only call other static methods and use static data in the class unless they have an instance of that class to use to call other methods or get to member variables.

You also notice that the word public appears twice in this code. The public keyword is a visibility modifier that basically says that what it modifies can be seen by code anywhere. Here we see it modifying the class first and the main method second. This says that this class can be used anywhere and that anything can call the main method. Java programs start in the main method, similar to C programs. In Java, main must have exactly the signature shown here. The only thing that can vary is the argument

name, “args”. Unlike C, every class in Java can have its own main method so a single program might have many ways that you can start it, each of which does something different. This can be very helpful for debugging because you can put main methods in helper classes that do nothing but run tests on that particular class.

The last main difference from C we see in this example is the fact that the method, main, is inside of the class, MyNumbers. In C, all functions are found at the top level, they have global scope. In Java, all functions are methods and must be placed in some class. There are no free standing functions in Java. The only things that exists at the top level in Java are classes and interfaces, which are similar to classes and will be discussed in depth later.

Also notice the comments in this code. There are two multi-line comments. These must start with /* and end with */, just like in C. You will notice that all the multi-line comments I have in this code start with /**. These types of comments are called documentation comments. As far as compiling goes, they are basic comments and are completely ignored. The documentation comments are used by a tool called javadoc that is part of any full Java development distribution which automatically generates documentation for your code. During the semester you will be turning in designs generated with javadoc. Javadoc comments and what should go in them will be discussed in more detail later on. For now, just know that your code should generally have a documentation comment directly above every class and every method in every class.

While this is example is simple, and shows some of the ways in which Java is like and unlike C, it is really a rather horrible example of object-orientation. The only object that we are using in this batch of code is System.out, and the way the example works, it isn't showing you much about using objects at all. This example makes heavy use of static, and static methods and data are much closer to the imperative style of C than to the object-oriented style you should be using in Java. Here is another code sample that puts objects to use in a much more significant way.

```
/**
 * Code Example 3.2
 * This class represents a wallet and has a main that you can run from
 * the command line.
 *
 * @author Mark Lewis
 */
```

```
public class Wallet {
	/**
	 * This is the main method for the MyNumbers class.
	 * @param args This is the array of command line arguments.
	 */
	public static void main(String[] args) {
		Wallet w=new Wallet(new Money(40.50),
			new ID("Mark Lewis","56569357"));
		System.out.println(w.getID().getName()+" has $"+w.getCash()+".");
	}

	/**
	 * This is the constructor for wallet that initializes the values
	 * in it.
	 * @param c This is the original cash in the wallet.
	 * @param id This is the id of the wallet's owner.
	 */
	public Wallet(Money c,ID id) {
		cash=c;
		idCard=id;
	}

	/**
	 * This method returns the cash that is in the wallet.
	 * @return The money object of cash.
	 */
	public Money getCash() { return cash; }

	/**
	 * This method returns the ID that is in the wallet.
	 * @return The ID object for this wallet.
	 */
	public ID getID() { return idCard; }

	private Money cash;
	private ID idCard;
}

/**
 * This class represents money.  It stores the dollars and cents separately as
 * ints.
 * @author Mark Lewis
 */
public class Money {
    /**
     * A constructor that takes a string and converts it to a double.
     * @param value The amount that it should start with.
     */
    public Money(String value) {
      setValue(Double.parseDouble(value));
    }

    /**
     * A constructor that takes a double and sets value with it.
     * @param value The amount that it should start with.
     */
    public Money(double value) {
      setValue(value);
    }

    /**
     * A constructor that takes dollars and cents as separate ints.  If cents is
     * bigger than 100 it moves the hundreds up to dollars.
     * @param d The nmber of dollars that it should start with.
     * @param c The nmber of cents that it should start with.
     */
    public Money(int d,int c) {
```

```
      dollars=d+c/100;
      cents=c%100;
    }

    /**
     * This method overrides the toString in Object and will return a
     * formatted string.
     */
    public String toString() {
        return dollars+"."+((cents<10)?"0":"")+cents;
    }

    /**
     * This private method sets the value from a double.  It is private so
     * that the class will remain immutable and it can't be called from the
     * outside, but having it as a separate method prevents redundant code in
     * two constructors above.
     * @param value The amount of money as a double.
     */
    private void setValue(double value) {
      dollars=(int)value;
      cents=(int)(100.0*(value-dollars));
    }

    private int dollars;
    private int cents;
}

/**
 * This class is a very basic representation of an ID card.
 * @author Mark Lewis
 */
public class ID {
    /**
     * This constructor sets up the ID card with the provided values.
     * @param n The name on the card.
     * @param num The number on the card.
     */
    public ID(String n,String num) {
        name=n;
        idNum=num;
    }

    /**
     * A getter function for the name.
     * @return The name on the card.
     */
    public String getName() { return name; }

    /**
     * A getter function for the number.
     * @return The number on the card.
     */
    public String getIDNumber() { return idNum; }

    private String name;
    private String idNum;
}
```

In real code you would find each of these classes in a separate file. In fact, Java would require it because only one public top level class can appear in a file. The reason for this is discussed later, but it has to do with requirements on file names. Notice that the only thing that is static in this code is the main in Wallet which creates a Wallet

object with money and an ID and then prints something about it.

This code is more object-oriented than the first example in large part because it uses objects. It also shows you how we construct normal classes and how we create objects from the classes. The three classes here are Wallet, Money, and ID. The Wallet class has two data members in it, one is of type Money and the other is of type ID. Both are private. In fact, you should notice that all data in this code is private. All data of public classes should be private. There are numerous reasons for this. One is that it allows you to limit the code that interacts with that data. For outside code to change the data, it has to go through a public function and the public function can do work, including checks of validity. For example, if you have a class for a grade the setter function might have checks to make sure the value of the grade was between -50 and 150 (some professors do give negative grades) and it will only set the grade if it is between those values. Also, it gives you more freedom to change implementations. We'll see that when we look at the Money class.

In addition to the main method and the two data members, the Wallet class also has three other methods in it. The first one is called Wallet and you will notice that it doesn't have any return type, not even void. Functions that have the same name as the class they are in are called constructors, and they are always called when we create a new object with the new operator. The use of new is seen in the main method. You simply follow new with the name of the class and an argument list as if you were calling a function. Those arguments are passed to the constructor. The other two methods in Wallet are simple get methods that return the data in the Wallet. Notice that this implementation does not allow you to actually change what Money object or what ID object are in the Wallet after the Wallet is created. This is an important fact to notice that we will discuss in some depth this semester.

After the Wallet class is the Money class. Money does not have a main in it here, though it certainly could. The money class has two data members, both are ints, that store the number of dollars and cents. You might wonder why I didn't just use a single double to store this. It turns out that floating point arithmetic is inexact and for that reason, it isn't very good to use with money. For example, 0.10 stored in a double will actually be 0.0999999999. However, I provide some functionality to deal with doubles and we could

provide functions to return doubles built from the ints without difficulty. This is an example of why we like to make the data members private. Imagine if you had written this class not knowing that it was bad to use doubles with money. It would need to store one data member that is a double and the functions would work with that. What if you made that data member public? Then you could have written code that directly uses that data member in other classes and if your code was being used by others, they might have written code that directly used that double data member. At the point when you learned that storing it as a double was bad, what would you do? You could try to change your class to use two ints as has been done here, but all the code that had used the double directly would have to be changed, even code that you didn't write. However, if you had made the double private and had provided accessors functions to get and set it, you could remove the double and store it as two ints, then change the logic in the accessor functions and all the code outside the class would continue to work. In fact, the Money class has a private method called setValue that could be used as a set method in this scheme.

To make this more explicit, here are two versions of Money. The first might have been the original implementation working with doubles and the second has been changed to use two ints. Notice that the interface, the way you call the method in the class, has not changed between the two.

```
/**
 * Code Example 3.3
 * This class represents money.  It stores the value as a double.
 * @author Mark Lewis
 */
public class Money {
    /**
     * A constructor that takes a double and sets value with it.
     * @param value The amount that it should start with.
     */
    public Money(double value) {
      setValue(value);
    }

    /**
     * This private method sets the value from a double.
     * @param v The amount of money as a double.
     */
    public void setValue(double v) {
      value=v;
    }

    /**
     * Returns the value of the money.
     * @return How much money there is as a double.
     */
```

```
    public double getValue() {
        return value;
    }

    private double value;
}

/**
 * This class represents money.  It stores the dollars and cents separately as
 * ints.
 * @author Mark Lewis
 */
public class Money {
    /**
     * A constructor that takes a double and sets value with it.
     * @param value The amount that it should start with.
     */
    public Money(double value) {
      setValue(value);
    }

    /**
     * This private method sets the value from a double.
     * @param v The amount of money as a double.
     */
    public void setValue(double v) {
      dollars=(int)value;
      cents=(int)(100.0*(value-dollars));
    }

    /**
     * Returns the value of the money.
     * @return How much money there is as a double.
     */
    public double getValue() {
        return dollars+cents/100.0;  // Using 100.0 converts to a double.
    }

    private int dollars;
    private int cents;
}
```

In this particular code example, because it is quite short and limited in functionality, the advantages of the two int method aren't significant, but if we had methods to add and remove money it would become more significant. The general rule here is that anything in the public interface can't be easily changed because there might be a lot of outside code that depends on it. For this reason, you want to limit what is public to a minimal, yet complete set, and never make data public because doing so limits your ability to change the implementations of classes.

Returning to the code in example 3.2, you should notice that there are actually 3 constructors for Money. The first takes a String, the second a double, and the third two ints. C would not allow this type of code because you can't have two functions with the same name. Java allows overloading, which is what we are using here. Overloading is when you have two methods that have the same name, but take different numbers or types

of arguments. Java figures out which version we are calling based on what we pass in the argument list. For example, `new Money(4.50)` would call the double constructor while `new Money("4.50")` would call the String version. In this case, both call the private method setValue(double). I wrote that as a separate method so that I didn't have to duplicate the code. Money also contains a method called toString(). As we will discuss later, it turns out that every class has a toString() method and it is called when the object has to be converted to a string. By default, it will return the class name and a unique identifier for the object. Notice that in the main of example 3.2 I use + between a string literal and a Money object. This implicitly calls the toString() method on the Money object and concatenates the strings.

The last class in example 3.2 is the ID class which is a simple container for two strings that represent the name and ID number of the person. I use a String for the ID number because many ID numbers have leading zeros. Something to note about the classes in example 3.2 is that they are all **immutable**. This means that after they are created, they can't be changed. In all three, the constructor sets the values in the private data and none of the public methods alter those values. This is a valuable property for many classes to have, and we will discuss the implications of it more when we talk about the String class, which happens to be immutable in Java.

There are a number of other significant differences between Java and C that aren't clear from these examples. One of the greatest strengths of the Java language comes from the vast libraries that it has. Learning how to read the API documentation at java.sun.com is a major aspect of this course. So far the only part of the library we have used is the System class that we used for output. This class is part of the java.lang package. We'll discuss packages in more detail in a few sections.

Unlike C, Java has no preprocessor directives. In C, those were the commands that started with # like #include and #define. Java has an import statement that looks like a #include, but it functions in a rather different way. #include actually copies a file's whole contents into the current file at that line. On the other hand import simply tells the compiler to look in certain places for things that it can't find otherwise. We'll discuss import more when we discuss packages.

We already mentioned that Java in not purely object-oriented because it has

primitive types. It does provide classes that are object wrappers for the primitives. These go by the names Integer, Double, Float, Character, Boolean, etc. Notice that they start with capital letters. All classes in the Java library start with a capital letter and yours should do the same. These classes provide a number of helpful functions. In the Money class in example 3.2 I used the static method Double.parseDouble(String) which takes a string and returns a double if the string can be converted to a double. If you want to do something with a primitive type you should look in those classes to see if the functionality is already provided for you.

## 4. Classes and their Structure

The examples in the previous section gave you a glimpse of some classes in Java and what goes inside of them. We should be more specific about the details of the contents of classes. A class can contain three things: methods, member data, and inner classes. Inner classes are a more advanced topic that will be covered later on. Data and methods can be modified with one of the three visibility keywords: public, private, or protected. They can also be modified with static and they can be modified with final as well. All three types of modifiers can be left off completely, but you should generally put a visibility modifier on all members of a class. Unlike C++, the bodies of all methods are found in the class that they are part of.

The three explicit visibilities and what they mean are as follows. Public implies that the member can be seen and used by anything, inside of the class or out. Private implies that the method or data can not be used by anything outside of that class. Protected says that the member can only be used by that class and subclasses of it. We'll talk more about subclasses when we talk about inheritance. This modifier is infrequently used, but critical when required so you will need to understand it later on. If you leave off all visibility modifiers the default value is called package private. This means that the member can only be seen and used by this class or other classes in the same package. Since nearly all of the code you write for this class will probably be in one package and because in general you don't have complete control over what goes inside of a package, this visibility isn't nearly as private as the name might indicate and should generally be avoided. For most things you put in a class, you will preceded them with either public or

private.

We have already seen static and discussed roughly what it means. Static can be applied to methods as it was in the main method or to member data. In both cases, it means that the member in question is associated with the class as a whole instead of with an object of that class type. Static methods and members are accessed by providing the name of the class followed by a dot and the name of the member. We saw this with System.out where out is a public static final member of the System class.

Static methods can only call other static methods or access static member data directly. To understand why this is, it helps to understand about **this**. Normal methods of a class have an extra parameter that is implicitly passed to them called "this". This is a reference to the object that the method was invokes on. Anytime we call a method or use member data in the current object, we implicitly use this. For example, the getValue() method of the second Money class in example 3.3 could be rewritten as such.

```
public double getValue() {
    return this.dollars+this.cents/100.0;
}
```

Not only does this alteration not change the meaning, it is basically what the compiler sees in the original version. One way of thinking of a static method is that it doesn't have a this. In the main of example 3.2 I had to create a Wallet object and call methods on it instead of directly using getCash() because the main method is not associated with any Wallet object implicitly.

It might seem unsafe making "out" a public member, but not only is it public and static, it also happens to be final. When the final keyword is applied to member data, it says that its value can never be changed. This is the one case in which it is safe to make member data public. You still can't change the implementation and that can be a drawback, but this usage is typically done with constants that either can't logically change types or, which have types that are unimportant to the outside code and changing the exact implementation won't break outside code as long as the ways in which it is used with the class they are in are properly modified.

Obviously, the job of a constructor in a class is to set up the values of member data elements in that class. For members that will be given the same value by all constructors, you can initialize them at the point of declaration much like you could

initialize a local variable in a function in C. It is worth noting that members declared in classes in Java will be automatically initialized to default values if you don't give them any value. For all numeric types the default is zero, for a boolean type the default is false, and for any reference to an object the default is null. Variables declared in functions, local variables, are not initialized, but Java will flag an error if you try to use one before you give it a value.

## 5. Primitives

Java has some differences from C when it comes to the primitive types though you might not notice those differences in normal usage, especially with Java 5.0 or newer. The integer primitive types in Java are byte, short, int, and long. These are all signed numbers stored in 1, 2, 4, and 8 bytes respectively. Notice that the long is a 64-bit int. As a result, it can store extremely large numbers in it and if you need exact arithmetic with integers larger than 4 billion, this is generally the way to go. If you get too large even a long will fail, and at that point, you can turn to the Java libraries and the java.math package which has classes for doing arbitrarily large arithmetic.

Java also has a char type for storing character data. A char is basically a 2 byte unsigned integer value and can be converted to an int. In C, the char type you worked with was a single byte that could store values between -128 and 127 (or 0 and 255 if you declared it unsigned) that were converted to characters using the ASCII encoding. Java uses Unicode instead. Unicode starts off like ASCII, but it contains more than 65,000 possible characters so that it can represent the alphabets of any language in the world. This won't impact you at all in your general programming, but it is definitely worth knowing, especially if you do get a job with a company that cares about internationalization.

Java has two floating point type: float and double. These types are basically equivalent to their C counterparts and use stand IEEE single and double precision arithmetic. A float is stored in 4 bytes and a double is stored in 8 bytes.

There is another type in Java that has no true parallel in C, that is the boolean type. In C, boolean values are handled as ints. In C99 (the 1999 standardized version of C) they introduced a type called bool, but it is really nothing more than an integer type where

0 is false and anything else is true. In Java, the boolean type stands on its own and has either the value "true" or "false". Expressions that take booleans in Java only work with booleans, not with integers. This has one wonderful advantage in that it eliminates one of the most common bugs programmers create in C. That is the bug where you leave out one = from a check for equality. For example, you type `if(i=5)` instead of `if(i==5)`. In C, both of these are perfectly valid code, though the first is inevitably a bug because not only does it overwrite i with the value 5, the code in the if always executes because the expression i=5 has a value of 5 which is true as a boolean. In Java, an if statement must be given a boolean expression so the expression i=5 will not compile.

As was mentioned earlier, the primitive types in Java are not objects. This decision was made in the name of efficiency because objects have a certain overhead and primitive types are built into the hardware and as such can run much faster. The downside is that there are some areas of Java that only work with objects. For this reason, Java has wrapper classes that can be used when you want treat a primitive as an object. For example, there is a class called Integer that will hold an int and a class called Double that will hold a double. A wrapper class exists for every primitive type. Remember that these classes not only wrap primitives, but also hold helpful functions that deal with the primitive types. Is you are ever wondering where you might find a certain helpful function that works with primitives, check the wrapper classes first.

The need for wrapper classes became less obvious with the introduction of Java 5.0 and a feature called autoboxing. Autoboxing is the automatic wrapping and unwrapping of primitives with wrapper classes. Because of this, you can write code that looks like you are using primitives as objects and it will compile. It is important to know that it is actually using wrapper classes as it could potentially impact efficiency.

## 6. Objects as References

One critical aspect that you must understand about Java is the handling of objects. In many ways, the handling of objects and primitives in Java is simpler than that in C. However, you have to understand how it is done and how it differs from C in order to be able to write working programs. The use of primitives in Java is virtually the same as in

C in that when you declare a local variable like “int i”, this creates a chunk of memory that holds an int and you can refer to it with the name “i”. Unlike in C, you can not get the address of this int and as a result, you can not pass primitive values by reference in Java.

Objects are exactly the opposite. When there is a declaration like “Wallet w”, this does not create an object of type Wallet. Instead, this creates a reference to a Wallet object which originally will not be valid. A reference in Java is very similar to a pointer in C only it is limited in what you can do with it for safety reasons. This improves the reliability of programs, but it also makes it so that you can't write system level code in Java. The main thing that Java disallows is pointer arithmetic. In C you could add to pointers to move them forward in memory or subtract from them to move backwards. The problem with this is that nothing checks to make sure you are moving to chunks of memory that you should be moving do. For that reason, Java does not allow it. Outside of system programming, this is not an issue.

In C, an invalid pointer could be set to have the value NULL. In Java, an invalid reference has the value null. Like booleans, pointers in C are basically integers storing an address and can be treated as such. In Java that might be the implementation, but the language will not let you treat references as integers. So a reference in Java will either have a value of null or it will have a valid reference to a valid object. C provides no such guarantee.

To get an actual object in Java you have to use the new operator. new is similar in many ways to malloc in C in that is provides you with a reference to a chunk of memory on the heap. However, new is far more type safe than malloc and it implicitly calls a constructor to initialize values. Example 3.2 used the new operator three times. Once each with Wallet, Money, and ID. The invocation of new returns a reference to the heap where memory has been set aside for the new object and it will have the type of the class name that follows the new keyword.

In C, every time you called malloc you also had to call free. Failure to do this would lead to a memory leak which would crash your program eventually. This is not a problem in Java because Java has automatic garbage collection. When you create an object in Java with new, a chunk of memory of the proper size is set aside for your use,

just as with malloc in C. Because Java doesn't allow things like pointer arithmetic and pointers aren't treated as integers, Java is able to keep track of what memory is currently in use, and when you stop using a chunk of memory it can clean it up in a process known as garbage collection. There are lots of ways of doing garbage collection and the technology behind it has gotten quite advanced. A full course could be taught on the topic, but for this course all you need to know is that you don't have to free your memory and you will never have a memory leak because Java does that work for you.

So when you declare a variable of an object type, you are actually declaring a reference to that type and you need to get an actual object through some other method, typically through a call to new. When you pass objects into functions you are also passing the references. One way to think of it is that Java always passes things by value, but that object variables are references so it is the reference that gets passed by value. This is significant in your programming because it means that if you pass an object that can be changed to a method, that method might change it. There is no way in Java to prevent that from happening. If you want to protect your original, you need to do what is called **defensive copying** and pass in a copy of your object. This is a benefit of immutable objects. Because they can't be changed, immutable objects can be passed around freely without having to worry about defensive copying.

In Java, arrays are also objects. We signify an array type by putting [] after any type and this produces a type that is an array of that type. Because the array type is a new type we can create multiple dimensional arrays by putting multiple sets of brackets after the type. The following code is a static method that could be put in the Money class. It is passed an array of doubles and returns an array of Money. The catch that it only keeps the dollars and throws away the fraction. To do this it declares two arrays, one for the whole dollars and one for the Money. Note that this is not the most efficient way to go, but I want to illustrate differences between arrays of primitives and arrays of objects.

```
/**
 * Example 6.1
 * Build an array of Money from doubles but only keep the Dollars.
 * @param vals An array of doubles.
 * @return An array of Money with vals dollars in each element.
 */
public static Money[] copyToMoney(double[] vals) {
    int[] dollars=new int[vals.length];
    Money[] ret=new Money[vals.length];
```

```
    for(int i=0; i<vals.length; ++i) {
        dollars[i]=(int)vals[i];  // cast to int not needed with autoboxing.
        ret[i]=new Money(dollars[i],0);
    }
    return ret;
}
```

There are a number of things to notice about this code that should stand out to you. Starting at the top, you see that we pass in just an array of doubles. In C you would need to pass in an array and a length. You can see why we don't need that in the declaration of the two arrays where we use vals.length. Because arrays are objects, they can have members in them. The one that you will use most frequently is the length data member which is public and final. Also, as you would expect with an object, we have to use new to create the object itself. The syntax is fairly straightforward where we do new followed by the type, followed by the size in brackets. What is stored in the array is basically the same as having the proper number of variables of that type. That's significant because an array of int actually stores the int while an array of object types stores only references to the objects. In both cases the arrays will start off initialized. Primitives take their default values (0 or false) while object references will be null. Because all the references are null to start with, we have to call new for each element of the array of Money objects in the loop. Notice that new isn't needed for the ints. The full implications of this will become clear later when we discuss inclusion polymorphism.

Beginning with Java 5.0 there is another way that we can write for loops over arrays or other containers. It doesn't help us with this example, but it can be very useful in many situations. This alternate form is commonly called the for-each loop. Consider the following code segment for taking the sum of the elements in vals.

```
int sum=0;
for(int v:vals) {
    sum+=v;
}
```

This code can be read to say, “for each v in vals” do the loop. Notice that we don't have to declare a loop variable that keeps the index, instead we simply give a name to the element of the array we are pulling out. This type of loop doesn't help for the code in example 6.1 because we need the index, i, inside of the loop. Any time that the index isn't needed inside the loop, the for-each format can be used quite effectively. This format would also be more efficient for certain types of containers. It doesn't really

matter for an array, but later in the semester when you learn about linked lists we will see that this format can be superior.

One other object type that is worth giving brief mention to is the String class. The String class is special in many ways. Unlike the primitive types, the String is an object type and there is a class to represent Strings in java.lang. As with other object types, variables of type String store only references. Unlike any other object though, the String class can be represented as a literal, for example “Hi Mom”, and the '+' has been overloaded to do string concatenation. These functions were provided for programmer convenience because they are things that people do a lot. To make it safe to pass String references around, the String class is immutable.

## 7. File Names and Naming Schemes

There are some interesting points to note about how Java forces you to name files and how it is recommended that you name things like classes, variables, and methods in your Java programs. The way it names files won't matter to you most of the time because you'll be “living” in an IDE that does many of these things for you, but it is still good to know about for several reasons.

In C, if you wrote a program that spanned multiple files, you generally wanted to use a makefile to organize things. You could call the files anything you wanted, no matter what code was inside of them. The makefile let you tell the compiler what pieces need to go together. Java forces you to name files the same as the one public class in the file with “.java” following the class name. The reason for this is simple, it means that the Java compiler always knows where to look for code and you never have to write makefiles (though there is a tool called Ant that is used for really large Java projects). Obviously, this means you can only have one public class in a file. In addition, if you change a class name, you have to change the name of the file too. The Eclipse IDE will do this for you if you refactor a class.

We'll talk about packages in Java later, but it turns out that packages in Java are represented simply by directories that the files go in. Here again, the advantage is that the compiler always knows where to look for something. Given a fully qualified package and class name, it tells the compiler the exact path to find the code or bytecode. More on

bytecode later as well.

When it comes to what you call the classes you write and the things that go in the classes, Java has about the same rules as C. You can't use keywords and you can't put most punctuation in identifiers because most punctuation means other things. Underscore is the primary exception. Names also can't start with numbers. Lastly, Java, like C, is case sensitive so “MyVariable” and “myvariable” are not the same. Naming two different variables by these names would be very poor coding practice because it makes code much harder to read an follow. For that reason, Java programmers typically follow some set naming conventions which I expect you to follow as well. Most things in Java are capitalized using the “camel” style. In this style, each new word begins with a capital letter. The name comes from the image you get of multiple “humps” in a long variable name. Whether the first letter is capitalized depends on the use of the identifier. Class names start with capital letters. Member data, methods, and local variables start with lower case letters. The one exception is static final data (basically constants), which are often named with all caps and underscores between the words.

I used this scheme in my earlier examples though it might not have been obvious. The whole Java API uses it as well and for that reason alone, you will find it is much easier to remember what you name things if you follow it too. Example class names might be “ThisIsMyClass”, “ClassHoldingData”, or “Money”. Member data, methods, and local variables would have names like “roundAndRound”, “loopCount”, “i”, or “cnt”. Constants that are static and final could have names like “PLAYER_DEAD” or “WEST”. Package names, by convention, are all lower case.

## 8. Scope and its Use

You all know about the scope of variables in C. Scoping in C was rather limited. You had global scope and local scope. Global scope went through the entire program. Local scope was through the entire block of code a variable was declared in. Functions could only have global scope. Variables could be global (though they shouldn't be unless you have a really good reason) or local. Most of the time you probably declared all your local variables at the top of a function and they had scope through the entire function. You could also have declared variables at the beginning of any block of code (right after

any open curly brace) and it would have a scope through that block of code. The scope of something is basically just the range over which it can be accessed.

Java has much more flexible scoping, though at this point you have only seen the beginning of that. What you have seen so far is that the global scope is only populated with classes. You never have functions or data variables at the global scope in Java. Things declared inside of classes have a scope in that class. This includes both the methods and the member data. In addition, you also have the same local scoping of variables in methods that you had in C functions, but with a bit more flexibility. Having extra scoping levels raises the question, what should be the scope of a variable, or equivalently, where should I declare a variable. The answer to that is fairly easy, variables should be declared in the narrowest scope possible. The reason for this general rule is that the broader the scope on a variable, the more places it can be misused or messed up. By limiting scope, when something goes wrong, you limit the number of places you have to check to figure out what went wrong.

If your code only has top level classes with data and methods inside of it, this leads to some simple rules. First, you only put data in the class if that same data is needed across several methods. If the data will be used by only a single method, it should be declared locally in that method. Using the same data name in multiple methods doesn't mean that variable should be in the class. For example, you might use the name "i" for all of your loop variables, but it should still be a local variable. The reason for this is that generally the value you give i in one method is not supposed to be remembered for use in another method.

Locally you should declare variables right before they are needed. This is in contrast to in C where you had to declare them at the tops of blocks. The advantage of declaring all variables at the beginning of a function is that you know where to go looking for them. However, good code should never have functions that are very long so this shouldn't matter. By declaring variables right before they are used, you limit their scope, which forces you to think about it when you try to use them in a different scope. If you find that the value is actually needed in that broader scope, you can change the location of the declaration. Many times you will realize that you don't really want the value there, you want something else and having the more limited scope will prevent a bug. Along

with this, if you use a variable inside of a loop and not outside of the loop, declare it in the loop. If it is something like the index variable in a for loop, then you should declare it in the first part of the loop.

We will also talk about inner classes this semester. These are classes that appear inside of other classes. Here again, the general rule of limiting scope can be applied. If a class is only needed inside of one other class, it should be made a private inner class so that it doesn't get altered in other places and so that the compiler will tell you if you accidentally try to do something with it outside of the class.

## 9. Packages

Packages were mentioned earlier in this document, but they deserve a bit of time to explain what they are and how they work. A package in Java is really nothing more than a way of organizing code. When you have groups of classes that are logically connected, those classes should be put in a package. The greatest advantage of this is that it prevents ambiguity in naming. For example, there is an interface (something similar to a class that we will discuss in the next section) called List in the package java.util. This interface is used by data structures like linked lists and array lists, things we will talk about this semester. There is also a class called List in java.awt that can be used to display a list in a GUI. Were it not for packages, having any two classes that have the same name would be a real problem. How would the compiler know which one you are referring to at any given time? The package construct fixes this problem by allowing us to refer to classes by the "full" names instead of just the class names. In fact, Java requires that you refer to classes by their full names unless you have provided an import statement to tell Java to look inside of a given package.

So using the example of the two classes called List, if we wanted to use the one that is related to a data structure, we could type in java.util.List. If we want to use the one that goes with a GUI, we could type java.awt.List. Obviously there is no conflict with these fully qualified names. However, it is a pain to always type the fully qualified names. In earlier examples you saw the usage of the class System. It happens that System is in the package java.lang so the fully qualified name is java.lang.System and we could have done a print with java.lang.System.out.println("Hello World!"). For anyone

who thought that just System.out.println was too long, this is obviously not a nice thing to type repeatedly. This is where import statements come in.

An import statement goes at the top of a Java source file and when it is used, the word import is followed by a fully specified class name or a package followed by ".*". Like any statement, an import ends with a semicolon. This looks like a #include in C, but what it does is very different. The import statement doesn't actually copy any code into a file like #include does, instead, it tells Java to go look in particular locations when it can't find classes. For example, if you put "import java.util.*;" at the top of your source file, then used List, it would be the java.util.List. Of course, if you import java.util.* and java.awt.* then there is an ambiguity again and you have to go back the the fully specified names.

One of the nice shortcuts of Eclipse is that when you use a class that is in a package that you haven't imported, it can help you quickly ad an import statement. After it has underlined the class name saying there is an error, you can move the cursor over the class name and hit Ctrl-Shift-M at the same time and it will automatically add an import statement for you. This prevents you from having to jump to the top of your file so much.

Not only are the libraries of Java arranged in packages, you can put your own code in packages. On your disk, packages are simply directories that have the same name as a package. So the code in java.util actually sits in a directory java/util under the compile directory and has the proper java source files in it. This follows in line with the fact that files have to be named after the classes in them. If the Java compiler knows the fully specified name of a class it knows exactly what directory to look in and what file to go to. If a class should be in a package, there should be a package statement at the top of the file for that class. For example, all of the code for this document I put in a package called edu.trinity.cs.ctoj. Code segments below will show the package statements for that. The project you will be working on for this class has code in the package edu.trinity.cs.gamecore. You will be importing from that package in every assignment this semester.

In Eclipse you can create packages by right clicking on a package and selecting New > Package. This will automatically create directories in your workspace. To follow standard Java naming conventions, package names should use only lowercase letters.

Beginning with Java 5.0, the import statement can also be used to import static values from various classes so that you don't have to type in the fully quantified names of constants.

## 10. UML Class Diagrams

UML stands for Unified Modeling Language and it is a standard on how to do drawings that communicate what is happening in code. There are many different types of UML diagrams and each one is intended to communicate a different aspect of code design or functionality. In this course we will only care about Class diagrams. These diagrams show the classes in a projects, what is in them, and ways in which they are related to one another. For now we won't worry about the ways they are related, instead we will just focus on the way the diagram shows what is in a class.

Figure 1 shows a standard UML display of the classes that we have seen in the examples so far. Each class is represented by a box divided into three regions. The top region simply shows the name of the class, the second region shows the data members of the class and the third region shows the methods of the class. Standard UML uses symbols that are easy to draw on whiteboards to give extra information about the things inside of a class. This is because part of the purpose of UML is to allow people to quickly and easily communicate ideas in programming and that can be done with sketches if everything in the pictorial language is easy to do by hand.

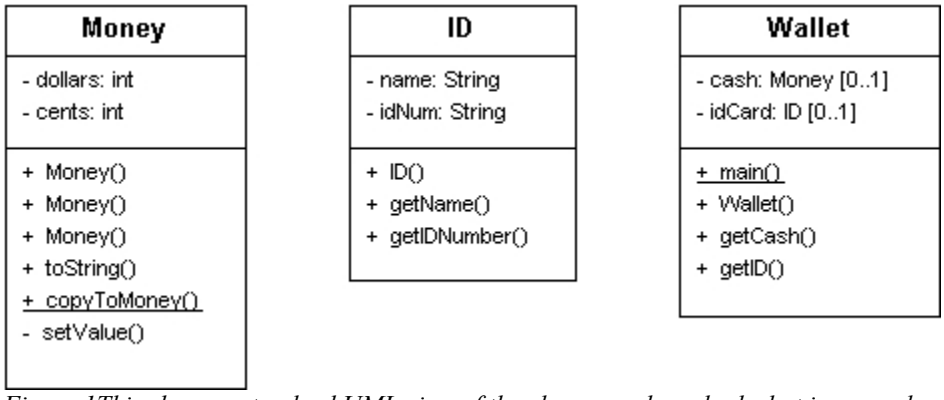


*Figure 1This shows a standard UML view of the classes we have looked at in examples above.*

Notice the UML notation is not exactly like code notation. The type follows the name after a semicolon instead of going before it as in C or Java. Before the member,

there is a symbol showing visibility: a - for private, a + for public, and though we don't see it here, a # would be used for protected. Also notice that static methods are underlined.

EclipseUML is a UML tool you could install with Eclipse. It has many options for how you can display things that go well beyond the normal UML standard. What is shown here is not the default way that EclipseUML will draw things, but it is good for you to know this style because it is what most people will be used to. EclipseUML allows you to specify what visibility items should be drawn. For many purposes you don't want people to see anything but public elements in a class and this is the default in EclipseUML. EclipseUML also includes the idea of a property. Much of the time, data elements have get and possibly set methods that you used to interact with them. These are referred to as properties of a class and by default EclipseUML will display properties instead of showing the individual get and set methods as well as the data. This makes the class representation shorter, but I think it can be confusing for beginning students so feel free to turn it off by going to Window > Preferences > UML > Class Diagram > Class and unchecking “Use property concept”.

Also, by default EclipseUML uses a “Eclipse” style of drawing where icons are used to show visibility. This type of representation is shown in figure 2. It looks a bit prettier and is just as easy to understand, but would be much harder to draw by hand. It doesn't matter to me which drawing style you use.

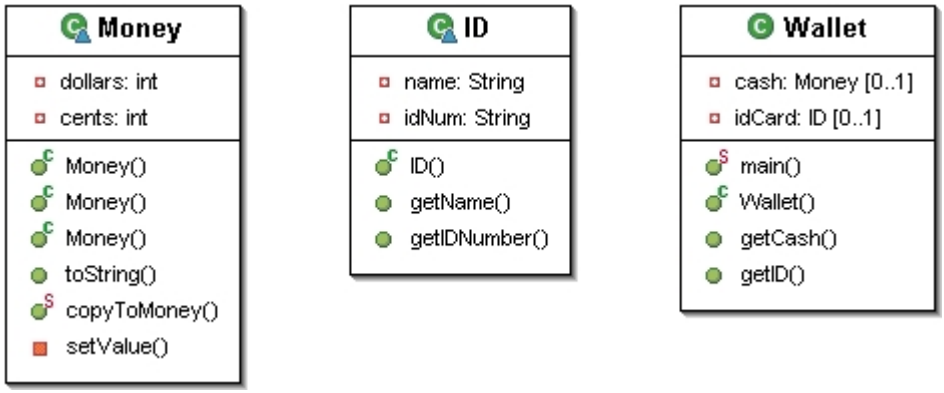


*Figure 2This is the same as figure 1, but with the Eclipse drawing style instead of the Standard UML drawing style.*

We will see more that can be added to UML diagrams at the end of the next section when we talk more about relationships between classes..

# 11. Inheritance, Polymorphism, and Interfaces

To this point we have covered a lot of details about the Java language, but nothing that really makes it seem like it is vastly different than C. You have to use classes and you can hide stuff by making it private, but that really isn't worth a whole course or the designation of a paradigm. Now we get into the topics that make designing and programming in Java vastly different from what you would have done in C.

We start with the concept of inheritance. Inheritance is a concept that is common to most class-based object-oriented languages. The name comes from the fact that it is used to allow a new class to "get" everything that was in an old class. By "get" I mean that all the data and the code that was associated with the existing class becomes effectively part of the new class. The new class can then add on extra stuff to extend the functionality. This usage of inheritance is what I typically refer to as **code reuse** because that was the major benefit that people saw in it. The new class, typically called the subclass or derived class, gains the ability to do everything the old class, called the superclass or base class, was able to do without copying the code over. This can be very helpful. As you have likely learned, one of the things we strive for when programming is to not duplicate code. A simple reason for this is that if you had an error in the original you now have multiple errors. Even if the original didn't have an error, but you want to change the functionality, you now have multiple locations where you have to change it. As I will describe later, this benefit also comes with costs.

It turns out that inheritance also provides another benefit in programming, one that wasn't part of the original motivation.[1] That benefit is **subtyping**. Formally we say that if a type B is a subtype of a type A, then any place that you want an A, you can use a B. Informally we say that B **is-a** A. In general, inheritance should only be used when we have an is-a relationship. The **has-a** relationship should be modeled with composition. That's when we put a member in a class like a car has-a steering wheel, so we would put an instance of steering wheel in the class for a car. The only place you see anything like inheritance in C is with numeric types of different sizes. So if a function wants you to pass an int you could always pass a short and that never causes a problem. However,

1 Subtyping wasn't originally part of the idea of inheritance because early object-oriented languages like Smalltalk have a more dynamic type system where subtyping isn't really an issue.

even then you don't get true subtyping. For example, a routine to sort ints won't work if you pass it an array of shorts. In Java, which is a statically typed, class-based, object-oriented language, when a class B inherits from a class A, B becomes a subtype of A and we can use an object of type B anytime an object of type A is expected. This ability is referred to a **inclusion polymorphism**. The word polymorphism literally means many shapes. In the context of programming it means many types. Polymorphic code is code that can work with many different types. Inclusion polymorphism is a form of true, or universal polymorphism which implies that the code can work with an infinite number of types. The other form of universal polymorphism is called parametric polymorphism and you will see it later with templates in C++ and possibly in functional languages like ML or O'Caml. The generics in Java also provide a style of parametric polymorphism.

The idea behind inclusion polymorphism is that we can create a base class and write functions that work with the base class, but that same code will also work with any derived class that you or anyone else writes. You might wonder why this is powerful if the derived class simply has all the methods from the base class. The real power of inheritance, as it turns out, lies not in the fact that you get methods from a parent class, but that you can override those methods to do something new. Methods that can be overridden are called **virtual methods** and in Java, all methods are virtual by default. When we call a method in Java, we get the version of the method closest to the type of the object we called it on. So if the class for the exact type of the object has the method implemented in it, then we will use the version in that class. If it does not, it will use a version in the direct superclass if it has one. This goes on up the inheritance hierarchy until an implementation is found. If no implementation is there then the code wouldn't compile.

To illustrate these concepts, lets look at an example. First we want to look at a UML class diagram of the classes. I'm using the Eclipse style drawing for this. Our superclass in this example is the class Shape. Our shape has two methods: area and draw. The area method returns a double that is the area of the shape and the draw method would be used to render the shape. The class stores a color for the drawing to be done in and has a get method for it.

This example includes two other classes that are subclasses of shape. Those are a Circle class and a Rectangle class. Both classes override the area and draw methods of Shape and then store the extra data they need to specify the shape. For the Circle class it adds a radius while the Rectangle adds a width and height. Notice that there is an arrow that connects from the subclasses the the superclass. This arrow has a filled in head on it. This type of arrow is used in UML to show that there is an inheritance relationship between two classes. In the diagram, the subclass points to the superclass. The reason behind pointing the arrow that way is that it reflects the knowledge that the classes have. The superclass knows nothing about the subclass. However, the subclass must know about the superclass. There are other arrows in UML as well. The most common is an open headed arrow that represents an association. This occurs when one class has a member that is an instance of another class. The association arrow points from the class that contains an instance to the class that member is an instance of. The reason for these arrows is that they show us relationships between classes. Any real program will have a number of arrow in it, but having too many arrows can be a bad thing, especially if they go all over the place. This is bad because it means we have a lot of dependencies in our code. Every arrow that comes into a class implies that something else depends on it and the changes to that class might break the code. EclipseUML can also add general dependency arrows that we will not discuss here. The code for these classes is shown below.

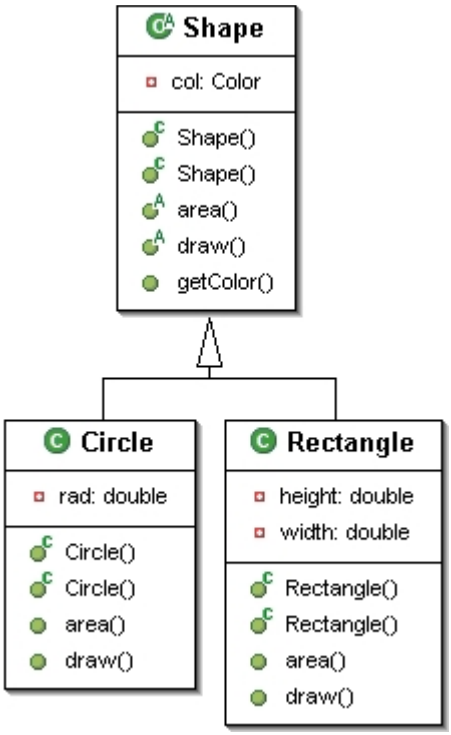


*Figure 3 - The class hierarchy for our shape example.*

```
/*
 * Example 11.1
 * Shape.java
 */
package edu.trinity.cs.ctoj;

import java.awt.Color;
import java.awt.Graphics;

/**
```

```
 * This is a simple example of a shape.  We will use it to explore
 * the concept of inheritance and subtyping.
 * @author Mark Lewis
 */
public abstract class Shape {
    /**
     * This is the default constructor and it will set the color to be black.
     */
    public Shape() {
        col=Color.black;
    }

    /**
     * This constructor sets the color of our shape.
     * @param c The color of the shape.
     */
    public Shape(Color c) {
        col=c;
    }

    /**
     * This method should calculate and return the area of the shape. It is
     * abstract because we have no idea how to do that for a general shape.
     * @return The area of the shape.
     */
    public abstract double area();

    /**
     * This method is supposed to draw the shape onto the provided
     * graphics object.
     * @param g The graphics object to draw it on.
     */
    public abstract void draw(Graphics g);

    /**
     * Return the color the shape should be drawn as.
     * @return The color the shape is drawn as.
     */
    public Color getColor() {
        return col;
    }

    private Color col;

}

/*
 * Circle.java
 */
package edu.trinity.cs.ctoj;

import java.awt.Color;
import java.awt.Graphics;

/**
 * This is a class to represent a circle.
 * @author Mark Lewis
 */
public class Circle extends Shape {
    /**
     * This constructs a circle with the given radius that is black.
     * @param r
     */
    public Circle(double r) {
        rad=r;
    }
```

```
    /**
     * This constructs a circle with the given radius and color.
     * @param r
     */
    public Circle(double r,Color c) {
        super(c);
        rad=r;
    }

    /**
     * This method returns the area of the circle.
     * @see edu.trinity.cs.ctwoj.Shape#area()
     */
    public double area() {
        return Math.PI*rad*rad;
    }

    /**
     * This method will draw a circle to the graphics object.
     * @see edu.trinity.cs.ctwoj.Shape#draw(java.awt.Graphics)
     */
    public void draw(Graphics g) {
        g.setColor(getColor());
        g.drawArc(0,0,(int)rad,(int)rad,0,360);
    }

    private double rad;
}

/*
 * Rectangle.java
 */
package edu.trinity.cs.ctoj;

import java.awt.Color;
import java.awt.Graphics;

/**
 * This class represents a rectangle and it inherits from our shape class.
 * @author Mark Lewis
 */
public class Rectangle extends Shape {
    /**
     * A constructor that sets width and height but leaves the color black.
     * @param w Rectangle width
     * @param h Rectangle height
     */
    public Rectangle(double w,double h) {
        width=w;
        height=h;
    }

    /**
     * A constructor that sets width, height, and color.
     * @param w Rectangle width
     * @param h Rectangle height
     * @param c Rectangle color
     */
    public Rectangle(double w,double h,Color c) {
        super(c);
        width=w;
        height=h;
    }

    /**
     * Standard area calculation for a rectangle.
     * @see edu.trinity.cs.ctoj.Shape#area()
```

```
     */
    public double area() {
        return width*height;
    }

    /**
     * Draw the rectangle. Don't worry about this yet. It will make sense later.
     * @see edu.trinity.cs.ctoj.Shape#draw(java.awt.Graphics)
     */
    public void draw(Graphics g) {
        g.setColor(getColor());
        g.drawRect(0,0,(int)width,(int)height);
    }

    private double width;
    private double height;
}
```

This code contains many details of things we haven't discussed yet. Many of those were present in the UML diagram as well. Starting with the Shape class we see the use of a new keyword: **abstract**. Both methods and classes can be labeled as abstract. When we say that a method is abstract, that means that it has no implementation. If a class contains any abstract methods, the class itself has to be abstract as well. You might wonder why we would ever want to have a method that doesn't have an implementation. The reason has to do with subtyping. In this example, we know that anything that is a shape will have an area. However, we have no idea how to calculate the area of a shape in general, so we really can't write a method for it. In the subclasses, like Circle and Rectangle, we know how to write the area method so we put it there. In fact, we have to put it there or else the Circle and Rectangle classes have to be declared abstract themselves. One significant factor about an abstract class is that it can not be instantiated. You can not create an instance of Shape using new with the code given here. The reason for this is simple, what would happen if you tried to call area on that object?

Aside from the abstract keyword, you should notice the package and import statements in the code. This code came from three different source files so it has three package statements and imports are duplicated. You can only have one package statement in a given file and there is no need to duplicate imports in a single file. Also notice the use of the word extends in the declaration of Circle and Rectangle. This is how we tell Java that those classes inherit from Shape. The word extends makes logical sense here because the subclass gets everything that was in the superclass and has the ability to add more to it as well. Hence it is an extension of the superclass.

Notice that each of the classes in example 11.1 has two constructors. In the case

of the Shape class, it has a **default constructor** as well as a constructor that takes a color. The default constructor does not take any arguments. As the name implies, it will be used by default, and it also exists by default. If you do not declare any constructors in a class, the class effectively gets a default constructor that does nothing. If you declare any other constructor, the default constructor will not be created by default. If you want the class to have a default constructor as well as other constructors, you have to explicitly declare the default constructor as I have done in Shape.

I said that the default constructor is called by default. The question is when this happens. Anytime we call new, we have to give an argument list, even if it is empty, so it really isn't defaulting to anything there, we specifically tell it what to use. Where the default constructor is implicitly called is when we instantiate an object of a subtype of that class. For example, look at the first constructor in Circle. This constructor takes a single argument that is the radius of the circle. However, a Circle is also a Shape so we need to call a constructor for Shape to setup those aspects of the Circle. In this case it has to set up the color of the Shape. If we don't tell it to do anything specific, it will call the default constructor of Shape before executing the constructor for Circle. If Shape didn't have a default constructor then this would be a compile error.

The second constructor for Circle shows how we can explicitly call a constructor for the superclass in Java, using the word super. This usage of the word super must be the first line in a constructor since the constructor for the superclass must be executed before the constructor for the subclass. In this case, we explicitly call the constructor that takes a Color object. Notice that this is the only way we could set the Color object in Shape. The member col in Shape is private and as such, not even subclasses have direct access to it. This is seen again in the draw methods in Circle and Rectangle where we have to call getColor() to get the color the shape should be drawn in. If col were made protected in the Shape class then the subclasses would be able to access it directly. Doing so would weakens the encapsulation a bit so we avoid using protected except when it is needed.

Super can be used in another way as well. When we are writing a method in a subclass, sometimes it is helpful to call a method in the superclass. However, if we have overridden that method and we try to call it directly, we get the version in the subclass.

This is most commonly a problem when we are writing the overriding method itself. Consider that class A has a method in it called func and that B is a subclass of A. In class B, we want func to do everything that it does in A, plus some other things. One way to do that would be to copy the code from A into B, then add the other stuff. You already know that's probably a bad idea because you don't want to copy code. Even worse, it might be impossible if we don't have the source code to A. To get around this, we can call the method on super, treating "super" as an object just like we can "this". So the definition in B might start with super.func() then do the other stuff it needs to do. Note that this usage of super doesn't have to be the first thing we do in a method.

Now it is time to look at what the subtyping of this inheritance relationship buys us. Example 11.2 shows a method that we could write called printShapeArea that we pass a Shape object to. This looks perfectly natural at first. However, consider the fact that Shape is abstract and you can't actually create an object explicitly of type Shape. So whatever we pass in is actually a subtype of shape. That also means we don't really know what method will be called when we call s.area(). It could be the version in Circle, the version in Rectangle, or a version in some subclass of Shape that we haven't even written yet. Which version of the method will be called isn't actually known until runtime. This type of calling is referred to as dynamic binding. This is the essence of polymorphism, the ability to write code that can use any number of types and will call methods we might not know about, but which we know will work because Java guarantees that any subtype object will have those methods.

```
/**
 * Example 11.2
 * This method will print the area of the given shape with formatting.
 * @param s The shape to print the area of.
 */
public static void printShapeArea(Shape s) {
    System.out.println("The area is "+s.area()+".");
}
```

We will see a lot more about this and the power that it provides us over the course of the semester. For now though you should think for a few seconds about how you would do this in C. What would you have to do to create this effect in C? Do you even know of a way to do this in C? If you do, is it robust and safe?

So you have now seen what inheritance is and have seen it used in a simple

example. Now we can get into some details of inheritance in Java. First, Java only allows single inheritance of classes. That means that a class can only have one superclass. The reason for this is that it prevents ambiguity. If you allow multiple inheritance then you can have a situation where a subclass inherits from two superclasses, and those two superclasses have members with the same name. If you call that method on an instance of the subclass, what actual method gets called? Even worse, you can produce inheritance hierarchies with "diamonds" in them. Image you have some superclass A which has two subclasses, B and C. That's fine. Now create a subclass D that inherits from both B and C. This causes problems galore. To see why, remember that when you inherit from another class, you get everything from that class. So an object of type B effectively has a full object of type A in it. The same is true for C. Now D has has an object of type B and an object of type C it in. That means it contains two objects of type A. When you make a call on D that comes from A, which version of A are you using? C++ allows multiple inheritance and adds a lot of complexity to get around these types of problems. The Java creators didn't want that complexity.

However, one can argue that there are times when multiple inheritance makes sense. For example, I am a professor and a Spurs fan. The class that represents me could be a subclass of both Professor and SpursFan. Java doesn't allow that with classes. Java does allow this type of functionality with minor limitations. Notice that all of the problems with multiple inheritance that were mentioned above only occur when there is ambiguity in members. That can happen with data or method implementations. It does not happen with abstract methods. Also note that in the example of me being both a professor and a Spurs fan, what we really want is the subtyping relationship, or at least that is what is most important. For these reasons, Java has the concept of an interface. An interface is like a class, but the only things that can go inside of it are abstract methods, static data, and inner classes. The inner classes we will talk about later.

We create an interface just like we create a class only we replace the keyword class with the keyword interface. Everything we put into an interface is public. Normally, interfaces are used to define exactly what their name implies, they give the public interface of a class, the set of methods that you can call. Because all the methods in them are abstract, there is no ambiguity. Also, you can't instantiate an interface just as

you can't instantiate an abstract class. The purpose of interfaces it to provide subtyping, nothing else. So with normal inheritance of classes, I said that you get both a code reuse functionality and subtyping. When you inherit from an interface, you get only the latter. As it turns out, this is generally a good thing because large inheritance hierarchies that make significant use of the code reuse have a tendency to be brittle and it becomes nearly impossible to make changes or upgrades to classes higher up in the hierarchy as those changes propagate through all the descendants of that class and often such a change will break at least one of them.

In our example of a Shape, the Shape superclass could have been changed into an interface if we hadn't wanted to put a color in them. In that situation, the class would have had no data and all the methods would have been abstract. If you ever have a class like that, you should probably change it to be an interface as that will give you more flexibility. In the code, we make a class inherit from an interface using the keyword implements much like extends was used. The implements keyword can be followed by a comma separated list of interfaces and there can be as many as you want. Just remember that you have to implement all the methods of all the interfaces or you will be forced to declare the class abstract. Lastly, you can make one interface a subtype of another interface and in that situation you go back to the keyword extends because the subinterface is extending what was defined in the superinterface.

The project makes extensive use of interfaces because most of the types are things that you will implement and which I could not know what implementation you would need. As a result, Screen, Block, GameEntity, and Player are all interfaces. In fact, Player inherits for GameEntity. Most of the work you will be doing on the project this semester is simply providing implementations for these interfaces and getting the various implementations to work together.

## 12. Generics

Beginning with Java 5.0 a new feature was added to the Java language. This feature, called generics, enhances type safety and provides some extra expressivity. If you happen to have any experience with C++, generics in Java look much like templates in C++ and serve much the same purpose. Generics provide a mechanism for **parametric polymorphism** in Java. They are typically used with pieces of code that can

work with any type. Prior to Java 5.0 code that could use any type was written so that it would work with Object because that is the base type for all object types in Java. Using this approach has the disadvantage that it requires a lot of type casting and it isn't type safe.

The most common example of generics comes with containers. A container class should normally be written so that it can contain any type. The java.util package provides a number of different container classes. Prior to generics, these classes were written to work with the Object class so they could hold anything. However, most of the time a given container should only hold one particular type and in this regard the non-generic implementations of these classes were not type safe. Because they worked with Object, you could put a String in, followed by a Double, followed by a Wallet. Generally this type of behavior indicates a programmer mistake and it will cause exceptions when the objects are taken out of the container. Using generics, the programmer can tell Java what type of objects are allowed in the container and Java can enforce that the programmer doesn't put other types in it. Example 12.1 shows code using a generic list of numbers.

```
/**
 * Example 12.1
 * This example code shows the use of a generic list and how it is type safe.
 * This function reads in numbers from a scanner and adds them to a list until
 * it reads a negative value.
 * @param sc This is a Scanner object that will be read from.
 * @return A list of Number objects that were read in from the Scanner.
 */
public List<Number> readIntoList(Scanner sc) {
    List<Number> ret=new LinkedList<Number>();
    int val=sc.nextInt();
    while(val>0) {
        ret.add(val);
        val=sc.nextInt();
    }
    return ret;
}
```

This example actually displays a number of different things that we have not seen before. The return type of the method is a list of numbers. The angle braces (less than and greater than) go around the generic type for the list. In this case, the list can only hold objects of type Number, which happens to be a supertype of the numeric wrapper classes like Integer and Double. List itself is an interface that is found in the java.util package. On the first line of the method we declare a List variable and create a LinkedList of Numbers for it to reference. LinkedList is a class that can be found in

java.util. Later on we will describe exactly what these are and you will write one of your own. As you should guess, LinkedList implements the List interface. By working with the interface you have more flexibility to easily change what implementation you are using later.

The argument passed into the method is an object of type Scanner. This is a class that was introduced in Java 5.0 to enable simple text input, something that had not been in Java previously because it is not used significantly in real programs. The Scanner class has handy methods for reading in different values easily. This piece of code uses the nextInt() method. As you might guess, this method reads in a string and tries to convert it to an integer. The loop continues until a non-positive value is found. All the positive numbers that are read are added to the list and then the list is returned. This code example also uses autoboxing. Technically the list doesn't store the type int, however, with autoboxing Java recognizes this and wraps the int inside of an Integer object and puts that on the list. This is valid to do because Integer is a subtype of Number.

It might not be clear how generics improve the code in this example over what we would have without generics. To illustrate this, imagine adding an extra line directly before the return that reads `ret.add("Hi");`. Adding this in the code above would prevent it from compiling and for good reason. We say that the list is a list of Number objects, but "Hi" is a String object. If we did not have generics, that line would compile just fine. The reason is that without generics, our list would work with any Object. Our String "Hi" is a perfectly valid Object so the compiler would have no problem. In fact, when we ran the code, the line adding "Hi" into the list would execute without issue. The problem would come later on when we were processing that list and expecting all the items in it to be of type Number. Without generics that code would crash when we pulled out the String object and tried to cast it to a Number. This might be in a very different part of the code from where the errant object was actually added. The true bug is the adding of the wrong object type. Generics help us to find this by producing syntax errors. Without generics this becomes a fairly difficult runtime error to track down.

This example only shows how we can use generic classes that were already written by other people and are part of the libraries. That is a very nice usage and it is the one that programmers will do the most on a regular basis. We also need to know how to

write our own code that is generic. There are actually two constructs we can make generic in Java: classes and methods. The LinkedList class is an example of a generic class. Later in the semester we will write a class called ListStack that will also need to be generic. The declaration for that might look like the following.

```
/**
 * Example 12.2
 * Part of a ListStack class showing how we declare a generic class.
 */
public class ListStack<E> {
    // Methods and data for the ListStack class using E as a type.
}
```

This class could then be declared much the same way as we did in example 12.1 by providing the type that we want to make a ListStack of. Notice that the placement of the angle braces in the declaration of the class is just like what we saw when we actually used the class. A class can also be made generic on multiple types, in which case the types are separated by commas inside the angle braces.

You might question the use of E as the type name in this example, especially after you have been told how important it is to use self-documenting coding style. This is actually part of the young Java standard for programming with generics. The names given to generics start with capital letters because they are the names of types. The use of single letters is standard because no one should ever be writing actual types (classes mainly) that have single letter names. So using a single letter name insures that you will not have name conflicts. I do not care if you stick to this particular standard. There might be times when you feel that a longer name that does more to document the purpose is helpful. E is the name given to a type for an element of something in all the generic containers in java.util. Since a Stack is a container and E is the type of the elements it can hold the name is fitting here as well.

In addition to making generic classes, we can also make generic methods. A generic method has its own generic types and can appear inside of a class that is or isn't generic itself. A simple example of this is a method that will fill a list with a particular number of occurrences of value.

```
/**
 * Example 12.3
 * This example shows a generic method that will fill the elements of a list.
 * @param <E> The list of element type.
 * @param lst The list to fill.
```

```
 * @param val The value to put in every element of the array.
 */
public static <E> void fillList(List<E> lst,E val,int num) {
    lst.clear();
    for(int i=0; i<num; ++i) lst.add(val);
}
```

There are a few differences to note between generic methods and generic classes. The generic type is specified before the return type in a generic method, not after the name of the method. More importantly, when the method is called, Java will infer the generic type. For classes we had to specify the type when the object was declared and instantiated. You can see this back in example 12.1 where we had to specifically state that our List could only contain Numbers. This specification was made three times on the first two lines of code in that example. When we use a generic method we don't normally have to do that. The reason is that the generic type typically appears in the usage of the function. For example, if I call `fillList` with a List<String> and a String, Java will figure out that E is a String. If you try to make the second argument something other than a String in that case, you will get a syntax error because Java will figure out that the types you are using aren't compatible.

Over the course of the semester we will write a fair number of generic classes both during the class meetings and as part of your project so there are a few other details of generics that we should discuss. Because of details of their implementation, generics have certain abilities and limitations that you wouldn't find with templates in C++ or other similar structures. The primary benefit you will find is that you can put stipulations on what classes can be used in a generic. In the examples above, the type specified in the generic could be any object type (class or interface). It is not possible to use primitive types with generics. If you want a list of int, you should declare a list of Integer and autoboxing will allow you to use the list with the int type without much of an issue. To be more specific, the examples above really allowed you to use any type that was a subtype of Object. Of course, that means you can only call methods that are part of the Object class inside of your generic class of generic method. Often you might want to allow a more specific range of types to be used. Particularly, you have the ability to specify that the generic type must be a subtype or supertype of a given type. This occurs very frequently in the code for the project as most of the classes and interfaces have the generic signature <B extends Block<B,E>,E extends GameEntity<B,E>>. This is actually a rather complex example of a generic because it not only specifies a supertype,

but the declarations have recursive definitions. It turns out that the recursive nature of the definitions will become transparent very quickly, but the part that matters here is the extends keyword. This tells Java that whatever type goes into the generic it must be some subtype of Block or GameEntity (depending on whether it is the first or second generic argument).

A simpler example of this would be if you wrote a class that needed to work with numeric values, but the user could decide if it should be integer or floating point values. The declaration of this class might look like the following:

```
/**
 * Example 12.4
 * A class that works with generics on a certain subtype.
 */
public class NumberCrunch<N extends Number> {
	// ...  Code for whatever the methods should do.
}
```

Given this declaration you could declare object variables of NumberCrunch<Double> or NumberCrunch<Integer> depending on what you were interested in doing. However, if you tried to declare NumberCrunch<String> you would get an error because String does not extend Number.

You can also use the keyword super in a generic declaration to say that a class or method can only be invoked with a generic type that is a supertype of a given type. This feature is much less used and we will not be making use of it in this course.

The rules for subtyping with generics are worth taking a look at. One might think that a List<String> should be a subtype of List<Object>. After all, everything that is a String is also an Object. However, Java does not recognize these as having a subtype relationship. The subtypes on generics are for the base types, not the generic types. The reason for this is quite simple to understand when you think back to the fact that all variables in Java are references. If we could put a reference to a List<String> in a variable of type List<Object>, the code would let us put any other type of object into that list. Here is what that would look like in code and why this isn't legal.

```
/**
 * Example 12.5
 * A segment of code showing why generic classes are subtyped on the base type
 * and not the generic type.
 */
List<String> ls=new LinkedList<String>();
ls.add("Hi Mom!");         // Perfectly valid addition of a string.
List<Object> lo=ls;        // Java won't allow this, but assume it did to see
                           // why it doesn't.
lo.add(new Integer(5));    // Big problem, this isn't a String.
```

The last line of code shows the real problem. The variable lo has a type List<Object> which would imply that it can hold any object, so adding the Integer to it would be fine. However, if List<String> is a subtype of List<Object> we create a situation where adding an Integer to a List<Object> causes problems. As a result, subtyping is only on the base types and even the base types only have a subtype relationship if the generic types are exactly the same. We'll see how to get around some of the issues this can cause below.

Before we get into the most nitty-gritty details, you should know about some limitations of generics. The information in generics only exists at compile time in Java, not at runtime. It is intended to provide an extra measure of type safety without impacting the running of the JVM so that people can use the new feature without all clients upgrading their JVMs. This has some side effects. For one, you can not instantiate objects of the generic type inside of generic classes or methods. Constructors are the one type of method that simply can't be virtual and a superclass can't specify what types of constructors the subclasses should have. As a result, Java won't let you do new on a generic type. The fact that generic information is compile time only also means that you can't make arrays of generic types. The reason for this is very similar to what was shown in example 12.5, but with the added issue that Java normally does runtime type checking anytime an object is placed in an array. Since generic information isn't available at runtime, this won't work properly. The solution for this was to completely disallow arrays of generic types. In practice this isn't a significant issue as programmers will use one of the subclasses of List instead of an array.

We saw above that List<String> is not a subtype of List<Object>. This has implications when we try to pass generic objects into methods. There are times when you really want to write a method that can work with a list of anything. More generally, there are times you can write a function and you don't care what the type in the generic is or you can specify a range of classes for it, but not exactly what it is. A simple example of this would be a class that is supposed to go through a List and print out all the elements. The question is, what should the type of the argument to that function be? The answer is List<?>. The question mark in this usage is a wildcard. It implies that any type could appear there and you don't care what it is. Along with that, it implies that the code in the function isn't going to depend at all on the nature of that type.

Suppose alternately that you wanted to write a function that would take a list of numbers and add all of them up and return the average. We might want to pass this function a List<Integer> or a List<Double> or even some other numeric type. If we declare it as one, it won't work for the others. There is a common supertype for the number types called Number, but List<Number> won't really work either because we don't have subtyping on the generic types. The proper type for the method is List<? extends Number>. Here we put a restriction on the wildcard so that it can only be classes that are subtypes of Number, but we still don't specify exactly the type that it will be.

As you might guess, not specifying the exact type does impose some restrictions. The primary one is that the object effectively becomes read only. More specifically, you can't call any methods that take the generic type as an argument. You can call functions that return the generic type, but what you get out will be either an Object or the type you specified in the extends clause. The wildcard can actually be used in places other than arguments to methods, though it isn't clear why you would want to. The only place you can't put a wildcard on a generic type is when you do new to instantiate a generic object.

## 13. Enumerations

Another feature that was added into Java 5.0 is that of enum types. The enum keyword exists in C and allows you to declare new types, but those types are effectively nothing more than alternate names for ints. This lack of type safety can lead to quite a few bugs in programs. For this reason, the enum was not originally put in the Java language. The point of an enum is to represent the elements of a small set of options. This comes up quite frequently in programming, so a type safe implementation of enumerations was added to Java with version 5.0.

The enum keyword defines a type in Java, just like the class and interface keywords do. Prior to having this, Java programmers would make a number of static final int variables for an enumeration. This method has the same pitfall as the C method because the values are ints and can easily lead to code where values are stored that aren't any of the allowed options. For example, if you had a class the represents a stop light, it might use an int to store the current color of light. The code for this might look like the following.

```
/**
 * Example 13.1
 * Using ints to make an enumeration.
 */
public class TrafficLight {
       // ... various code
       private int currentColor=RED;

       public static final int RED=0;
       public static final int GREEN=1;
       public static final int YELLOW=2;
}
```

The code in the class would check the currentColor variable to see what state the light was in. The capitalization isn't required, but many people us it to denote constants in code. The problem is that you can assign a value to that variable that isn't in the range [0, 2]. What should the code do if currentColor has a value of 10? What is needed is a construct that can only have the three values that are allowed. To do this we could use the simplest format of an enumeration. Our code would change to look like this.

```
/**
 * Example 13.2
 * Switching the code from the previous example to use an enum.
 */
public class TrafficLight {
       // ... various code
    private LightColors currentColor=LightColors.RED;
    public enum LightColors { RED, GREEN, YELLOW }
}
```

With this, our enumeration is completely type safe. The only things that have the type LightColors are RED, GREEN, and YELLOW. There is no way you can put an invalid value in the variable currentColor. The one complication is that an enum makes an object type so currentColor would be null if it had not been initialized.

Java effectively creates classes for an enum that are subclasses of java.lang.Enum. Because these are objects they can have a number of nice methods. Java also makes sure that those classes have nice properties. For example, there will only be one RED object ever created in the code above so you can use == to check equality. Some helpful methods that are defined for all enum types are toString(), valueOf(), and ordinal(). Enumerated types also implement both Serializable and Comparable. Those things might not matter to you explicitly in this class, but if you use enum types they will inevitably help you out at some point.

Code that uses enums typically has switch statements in it. Prior to Java 5.0, a switch statement could only be made on integer types. With the introduction of enum a switch can also be done on enumerated types. Our TrafficLight class might contain a

method with the following code to tell a car what to do as it approaches the light.

```
/**
 * Example 13.3
 * This method would go in the TrafficLight class of example 13.2 and would
 * tell an approaching car what it should do in response to the light.
 */
public void approachLight(Car car) {
       switch(currentColor) {
       case RED:
              car.stop();
              break;
       case GREEN:
              car.proceed();
              break;
       case YELLOW:
              car.slow();
              break;
       }
}
```

Notice that in the switch statement we don't have to put LightColors.RED. In fact, it is an error to do so. Java will figure out from the type of the argument given to switch what enumeration the values should be pulled from.

Given that the enumerated types are actually classes, you might expect that other things can be added to them. This is indeed the case. Each element of an enumeration can have values attached to it and each one can even be given different implementations of methods as you would get from inheritance. We aren't going to go into the details of how to do that, but you should be aware that it is possible and could be helpful for certain uses of enum.

## 14. Exceptions

This is a topic that we will cover more completely late in the semester, but you need a basic understanding of exceptions to do almost any Java programming, even if just to understand the error messages that you get. Exceptions are the recommended way of signaling errors in Java. As we will discuss later in the semester, they have many advantages over other methods of signaling errors. For now though you need only know that they are a way of dealing with errors in programs, and have some idea of the syntax that goes into a program to deal with them.

If you write code that might throw an exception that you want to handle, or in some cases that you must handle, that code should go inside of a try block. After the try block you put one or more catch blocks that handle different problems that might arise in the try block. Example 14.1 shows a method that includes a try block as well as a catch

block.

```
/**
 * Example 14.1
 * This is a method to demonstrate the use of try and catch.
 */
public int checkInt(JTextField field) {
	try {
		value=Integer.parseInt(field.getText());
	}
	catch(NumberFormatException e) {
		JOptionPane.showMessageDialog(field,"You must enter a numbers.");
	}
}
```

This simple method could go inside of a class that has a member called data and shows some type of GUI where the user is supposed to enter in a number in a text field. The parseInt method in the class Integer can throw a NumberFormatException if the String that is passed to it is not a valid integer. So in this case, if the user input "abc" and then did whatever caused this function to be called, the program would display a dialog box informing the user that they were supposed to input a number instead of what they had typed in. In this case, the try and catch aren't required, but there are some methods that you can call, which to require you to have a try and catch in place and Java will report an error if they aren't there.

## 15. Inner Classes

Starting with Java 1.1, the ability was added to have classes at scopes other than the global scope. Normally we think of these as classes inside of other classes, but they are actually more flexible than that and they have more power as well. Classes that are not at the global level are called inner classes.

Inner classes come in a number of different flavors. Which one you use at a given point in time depends on what your needs are. We start with the most straight forward type of inner class, a class that is written inside of another class. Code wise this looks exactly like what the description implies, we write a class like normal only we put the code for it inside of another class. Like anything else we put in a class, these inner classes can be modified with a visibility and can be static or non-static. If code outside of the outer class needs to use the inner class then it should be public. Otherwise it should probably be private. The question of whether an inner class should be static can be

answered with a double negative statement saying that if it doesn't need to be non-static, then it should be static. Basically, you should make inner classes static by default.

So why would you make an inner class non-static? To answer that you need to know a bit more about the abilities of inner classes. Not only is an inner class in Java scoped inside of the outer class, implying the name has to be specified further by outside code, inner classes, like methods in classes, have access to the private data of the outer class. So methods in an inner class have access to private stuff in the outer class in the exact same way that methods in the outer class do. Of course, the methods of the inner class also have access to private data in the inner class as well. Similarly, just as a non-static method knows about the `this` object by default, a non-static inner class also knows about the instance of the outer class that instantiated it. So by default, methods of a static inner class can get to private static elements of the outer class while non-static inner classes also get default access to the private non-static elements of the outer class for the instance that created the inner class object.

If that seems confusing, a simple way to think of it is this, if the inner class needs to use non-static data in the outer class, it shouldn't be static. Since most data in classes is not static, a less correct rule would be simply that if the inner class doesn't need access to the data in the outer class then the inner class should be static and if it does need access it should be non-static.

An example of this style of inner class that we will see this semester is a node class in a linked list. A piece of code showing what this looks like for a linked list based stack can be found in example 15.1. We won't worry about the details of the functionality of the code here. Instead, we want to focus on some details of the inner class and how it is used.

```
/*
 * Example 15.1
 * Created on Jan 18, 2005
 */
package edu.trinity.cs.ctoj;

/**
 * This is a simple implementation of a linked list based stack.
 * @author Mark Lewis
 */
public class LinkedListStack {
    public void push(Object data) {
        head=new Node(data,head);
    }
```

```
    public Object pop() {
        Object ret=head.data;
        head=head.next;
        return ret;
    }

    public boolean isEmpty() {
        return head==null;
    }

    private Node head;

    private static class Node {
        public Node(Object d,Node n) {
            data=d;
            next=n;
        }
        private Object data;
        private Node next;
    }
}
```

In this case, the inner class functions more like a struct than a class. Because Node functions like a struct, it really has no functions in it and therefore our rule above says that we should make it static. If for some reason we added something to the nodes such that they were supposed to read or modify the head pointer in a list, then the Node inner class would need to be non-static. That is not the case here.

For an example of a very different type of use of inner classes, look in the java.awt.geom package. In that package many of the top level classes are abstract. This includes Rectangle2D. They have public inner classes in them that inherit from the outer class and fill in the methods so that they are not abstract. This design might not seem intuitive at first, but it is quite nice once you get used to it.

There are two other types of inner classes in addition to the standard named inner class that we have looked at. These other types of inner classes should be used for different types of situations. The first alternate type provides a narrower scope for an inner class. In the code for example 15.1, the class Node has a scope through the entire class LinkedListStack. In this case that is needed because it is used in more than one method in that class. What if an inner class were only going to be used in one method? Following the rules stated much earlier about limiting scope, one might feel we should have the inner class limited to the scope of just the method it is used in. This is indeed possible, and it is called a local inner class. To make a local inner class, we need do nothing more than declare a class inside of the method that we will use it in.

In this case, there is no need for visibility modifiers or a static modifier. The class

only exists inside of the method so the idea of visibility doesn't make sense, it is by default private to the method. Whether or not it is static is taken from the modifier of the method. If the class exists in a static method then it will be static, otherwise it won't. Personally I have never used a local inner class though I have written some chunks of code where it could have been used.

The reason I have never used a local inner class is that I typically use a close relative of it instead: the anonymous inner class. Obviously you only use a local inner class when something will be used over a very limited scope. If you will only be instantiating variables of that type on one line then it turns out you really don't even need to give the class type a name as long as it can be referred to by some existing type. So as the name implies, anonymous inner classes are inner classes that you don't give a name to. This might seem like something that you would rarely do, but starting with Java 1.1 the libraries added a number of places where it is helpful, especially with GUIs. You use the anonymous inner classes when you need to write a short class that has only one or two methods. The advantage is that you can write it directly into the statement of code that needs it. Example 15.2 shows a method that demonstrates the use of two anonymous inner classes. The first is set up to do something when a button in a GUI is clicked and the second is used to make a sort algorithm sort some strings without considering case.

```
/**
 * Example 15.2
 * This method demonstrates the use of anonymous inner classes.
 */
public void aMethod() {
    JButton button=new JButton("Exit");
    button.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) { System.exit(0); }
    });

    String[] str={"This","is","a","test","Sort","mE"};
    Arrays.sort(str,new Comparator() {
        public int compare(Object o1,Object o2) {
            return ((String)o1).compareToIgnoreCase((String)o2);
        }
    });
}
```

The first use of an anonymous inner class here would add a listener to a button. This type of usage was one of the major motivators for adding inner classes to the Java language because this became the proper method of handling GUI events in Java 1.1. With the listeners, the code inevitably includes a large number of small classes. The

anonymous inner classes allow you to write those small classes into the statements where they are needed so you don't have to jump to a different part of code. Notice the syntax that is used. We call new and give it a class or interface name. This will be the supertype of the class we are creating. In the first usage here it is the interface ActionListener. This almost seems to go against the fact that you can't instantiate an interface or an abstract class. However, you aren't instantiating an ActionListener here, you are instantiating a subclass of ActionListener and that subclass has to implement all the methods of the interface. In this case, the only method is actionPerformed. So our new type doesn't have a name of its own, but we can, and will, treat it as an ActionListener. Where the anonymous inner class differs from a normal instantiation is that after the close parentheses, we put a curly bracket and have the definition of the rest of the class inside of it.

The second anonymous inner class in this example is a subclass of the interface Comparator, which is used to do polymorphic sorting. In this case, we have written a comparator that will cause the sort to happen in a case insensitive way. Both of these usages are very simple, but that is the standard for an anonymous inner class. If the class were going to be more complex we would typically use some form of named class, probably a normal inner class. Like the local classes, the anonymous inner class is static or not depending on what method it is in and visibility is not an issue since there is no name to refer to the type by. Also keep in mind that because the local class and anonymous inner classes are inner classes, they have access to private aspects of the outer class. This means that if the code for one would be long, you can put most of the code in a private method in the outer class and have a method of the inner class call it.

With local inner classes and anonymous inner classes there is another aspect that is worth considering. It has been mentioned that these constructs have access to the private data and methods in the outer class. What about local variables though? After all, these classes are written inside of methods and there can be variables declared in those methods above where the classes are written. In example 15.2 we have the variables button and str. Our anonymous inner classes didn't need these variables, but what if they had? There are cases when you might want to refer to a local variable inside of a local or anonymous inner class. It turns out that you can't simply refer to any local

variable when you write the class. After all, the instance of the class could exist long after the method has terminated. However, a local or anonymous class can refer to any final variable declared in the scope that the class definition sits in. Final variables have to be given a value at declaration and can't be changed after that. You can make any variable final by simply putting the final keyword in front of the type. Note that parameters to methods can also be final.

It is worth noting that final, when applied to a variable or member in Java, has the meaning that the memory cell for it will never change in that scope. When the variable or member is an object type, the object itself can change, but the reference can not. This is a significant distinction from the use of const with references in C++. Also note that final applied to a variable or member means something very different from final applied to a class or method. As was mentioned above, a final class can't be extended by any subclasses. Similarly, a final method can't be overridden by any subclasses. The similarity is that a final class or method will always be what is defined there and the functionality never can change. The details of the two uses of final are quire different though.

## 16. Compiling, Bytecode, and the Java Virtual Machine

One last topic that should be discussed about Java to give you a full understanding is the way that programs are compiled and run. In C, when you ran the gcc compiler, it took the source code, ran it through a preprocessor, then a compiler, and finally a linker made a file that was in machine executable format. That file could run on the platform it was compiled on, but generally no other platform would run it without some translation software. That is to say that if you compiled under Linux on an x86 machine, you couldn't run it under Windows or any other OS, nor could you run it on any other chip architecture, even if it was under Linux. The size of the file that is generated really depends on the compiler and the platform, but typically they are fairly large, larger than your source files.

Java does not generally compile to machine executable format. The reason for this is mostly historical, but there are practice reasons it has stayed that way. When Java was originally developed, it was intended to be a language for use in small devices like

set-top boxes or cell phones (though cell phones weren't viewed as a huge market in the early 90s). For this reason, Java programs were supposed to compile to something that was small so that you could put it on machines that didn't have the resources of a full PC. In addition, they wanted the files to be non-platform specific. So that the same program could be easily moved from one device to another, even if the devices had different operating systems and different processors. When the web became big, it also became a nice way to send programs across the net to run client side. That wasn't the original market, but it was what make Java a huge success. To do these things, they made Java compile to a bytecode that was small and not specifically suited for any real machine. In order to make the bytecode run on a machine, you have to have a **virtual machine** on the platform that you intend to run on. The virtual machine would be a full executable for that platform and would be platform specific, but you only had to write one for each platform and then all the bytecode for any program could be shared between platforms easily.

So after you write a program in Java, you compile it much like you compile a C program. Eclipse will do this for you, but it uses the javac command that is part of the Java Developers Kit (JDK) and can be freely downloaded. This produces a number of files ending with ".class", one for each class in the program. These files are in bytecode. To run the program you use the java command followed by the name of the class that contains the main you want to run. Again, this is hidden from you in Eclipse. The java program comes with a JDK or with a Java Runtime Environment (JRE).

It is worth noting that the use of bytecode and a virtual machine caused some problems for early version of Java. In particular, running something other than native code in an interpreter slows things down. The original implementation of Java were quite slow compared to C and C++. However, the success of Java moved a lot of research interest into how to fix that problem and beginning with version 1.3 and 1.4 Java became quite competitive with languages that produce native code in mosts tasks. A big part of this was due to the development of **Just In Time** compiles (JITs). These are virtual machines that compile the bytecode to machine code on the fly right before execution. Newer versions of JITs, like the HotSpot compiler that is part of the Sun JRE, do significant optimizations on the parts of code that execute most frequently and as a result

they can produce very efficient code without the user ever knowing that they create the machine code when you do the execution.

## 17. String Processing

We have already seen that Java represents strings by a class called String. This class is significantly more robust than the char arrays that you used for string data in C. Some of the significant aspects of the String class have already been mentioned, but they bear repeating. Strings are not primitives, they are objects instantiated from a class. Strings are immutable. This means that when you create a String object, it will never change. This has the advantage that you never have to do defensive copying. However, when you do things that would seem to modify a String object, they actually create a new String object. Examples of this are methods like toLowerCase and replace. When these are invoked, the original String object remains unaltered and a new String object with the specified changes is created. Because String is immutable, the default constructor and copy constructor are rather unnecessary. They are in the library because they were added early on before all the implications of immutability were understood. This type of oversight occurs in a number of places in the early Java libraries.

One critical fact about immutability that is often misunderstood is that it is the instantiated objects that are immutable, not the references to them. If you declare a variable and make it point to a String object, you can later make it point to a different String object simply by doing an assignment statement. You could prevent that by making the variable final if you wanted. Also, immutable doesn't mean that the class itself doesn't change. Again final can be used to say that a class can't be extended and the method implementations can't change. Immutable isn't really part of the Java language. There is no keyword that says a class is immutable and that the objects instantiated from it can't be changed. The writer of the class must be careful to make there class immutable. An immutable class will have no set methods, nor any other methods that change the data in it after the constructor is called. Also, a truly immutable class should be final as well because otherwise someone could write a subclass that can be used in place of your immutable class that is not immutable. This could cause problems if you write code that assumes immutability.

When you are coding you will frequently use string literals with the double quotes syntax. Java produces true String objects for these. So the expression “Hi Mom!”.length() actually makes sense in Java. This would not make sense in C where the string literals are translated to arrays of characters. Like C, you can put escape sequences in your string literals to represents characters that you can't normally type. For example, \n will become a newline and \t will become a tab. If you want to put a double quotes in a string use \”. Because the backslash is used for escape characters, to get a backslash in a string use \\. As with C, single quotes are used to represent character literals. So 'a' is of type char while “a” is of type String.

The String class is the only class that has any operator overloading in Java. The creators of Java understood that string concatenation is such a common activity that being able to to it with a '+' would be a great benefit to programmers. You need to be cognizant of the fact that concatenating two strings will produce a third string. Repeated concatenation of small strings to build a large string is extremely inefficient for this reason. Unlike '+', '==' is not overloaded for String or any other object type. Doing '==' with a String or any other object type will check if two references refer to the same object. It is like doing '==' on pointers in C or C++. Generally in Java, if you want to check if two objects are equal you should use the equals method. The following example shows this.

```
/**
 * Example 17.1
 * This main method demonstrates how comparing strings with == might not
 * produce the results that you like.  You should use .equals instead.  That
 * is true for all objects.
 */
public static void main(String[] args) {
       String s1=”test1”;
       String s2=”test”+1;
       // At this point both strings contain “test1”.
       if(s1==s2) {  // This won't be true.
              System.out.println(“The strings are the same string.”);
       }
       if(s1.equals(s2)) {  // This will be true.
              System.out.println(“The strings have the same value.”);
       }
}
```

Like the wrapper classes for primitive types, the String class also contains a number of helpful methods that you can use for processing String objects. Because a String in Java is not an array of characters, you can not use the square brackets after a string to get hold of a character at a particular position. Instead, you do this with the

charAt(int) method. The index that you pass in is zero referenced, just like for an array. To get a larger section of a string you can use the substring methods of the String class. These take either a starting index or a starting and ending index. The single argument version returns a String that contains everything from that index on. The second form returns a String that contains from the first index up to, but not including, the last index. Other helpful methods include indexOf, startsWith, endsWith, toLowerCase, to UpperCase, and trim. The last three might seem to alter the String object. This is not the case though. Instead, they return a new String object with the proper alterations. You should note that many of the function in the String class return a String object. This is required since the class itself is immutable.

Imagine you had a function that was going to read one character at a time and build a String from it. A simple way to write the code for this would be to just concatenate the new character onto the end of the string repeatedly as in the code segment below.

```
/**
 * Example 17.2
 * One way you could build a string one character at a time.  This is very
 * inefficient.  Don't worry about the details of the Reader class at this
 * point other than it can be used to read characters.
 */
public String readString(Reader in) {
	String ret="";
	int c=in.read();  // returns an int so it can single end of input.
	while(c>=0) {
		ret+=(char)c; // The int is cast to a character to append.
		c=in.read();
	}
	return ret;
}
```

This code will work just fine, but it is extremely inefficient. The reason goes back to immutability. The line ret+=(char)c; must build a new String object each time it is called. Making that String object will require copying all the characters from the old String object over and adding on new one. So the number of characters copied grows by one each time through the loop. This behavior leads to $O(n^2)$ performance both for time and memory usage. That is extremely inefficient for something that should go much faster.

The way to improve on this is to use the StringBuffer class. The StringBuffer class is not a String type, but it is easily convertible to String and it is not immutable so you can efficiently build up a StringBuffer, then convert it to a String. The example below shows the same function, but this time using a StringBuffer. This is how you

would actually want to write this function.

```
/**
 * Example 17.3
 * Building a string one character at a time with a StringBuffer for efficiency.
 */
public String readString(Reader in) {
       StringBuffer ret=new StringBuffer;
       int c=in.read();  // returns an int so it can single end of input.
       while(c>=0) {
              ret.append((char)c); // The int is cast to a character to append.
              c=in.read();
       }
       return new String(ret);
}
```

Inevitably you could write this same function using an array of chars, but that won't grow as easily, you will have to include code to constantly make bigger arrays. With StringBuffer, that work is done for you.

Beginning with Java 1.4, the Java libraries have support for regular expressions. This appears in several of the methods in the String class: matches, replaceAll, replaceFirst, and split. Regular expressions are patterns of characters that have fairly simple rules. Their power is limited in many ways, but they can be remarkably useful because even with their limited power, the code it would take to do what they can do is often quite significant. For a full description of regular expressions in Java see the java.lang.regex package. In particular, the Pattern class has significant documentation at the top of it telling you about valid regular expressions in Java. A simple regular expression that can be quite helpful is " +", note there is a space in front of the plus sign. This represents one or more spaces. When you want to split a line of text on spaces you can pass this string to the split command. If you only give it a single space then it won't properly split it up when there are two or more spaces between words.

## 18. Arrays

We first talked about arrays in section 6. They are the most common way of grouping things together in programming languages. An array is basically a collection of the same type of object that can be referenced by an integer index. Arrays in Java are very different from their counterparts in C. In C, an array is basically a pointer to a chunk of memory that is large enough to hold the specified number of the specified type. For example, an array of 10 ints will point to a chunk of memory 40 bytes in size. Because the array is basically a pointer, any functions working with an array had to be passed both

the array and the size of the array because arrays don't know how big they are implicitly.

In Java, arrays are objects. This has quite a few implications for your programming. The most immediately significant one is that being an object means it can have extra data associated with it. Every array object can tell you how many elements it holds by its length member. This prevents you from having to pass the length to functions, just pass the array. Good Java code will use the length in the objects instead of constants or other variables as that length is always right. We declare array type in Java by simply adding [] after the type we want. You have already seen this with the String type in the argument for main. Java will actually let you put the brackets after the type or after the variable. I recommend putting it after the type as that makes the meaning more clear. For example, int[] and String[] are types in and of themselves, and when you declare a variable of that type it is a reference that can only point to that type of object.

Like all other object variables in Java, the declaration of an array variable is actually a declaration of a reference. In order to use this reference we need to make it point to something. Much of the time we will do this with new. The following example shows the declaration of an array of ints and how we could fill that array with values.

```
/**
 * Example 18.1
 * Creating and filling an array of integers.  We will return that array just
 * for good measure.  The values put in the array count backwards from the
 * specified size to 1.
 * @param size How big to make the array.
 * @return Ann array of the specified size filled in reverse order.
 */
public int[] fillIntArray(int size) {
	int[] ret=new int[size];
	for(int i=0; i<ret.length; ++i) {
		ret[i]=size-i;
	}
	return ret;
}
```

Notice that the loop uses the length of the array to tell it when to stop and not the variable that was passed in. In this code they are equivalent, but if we altered the code slightly that might not be true and such an alteration would cause our code to either crash or not initialize some of the values.

You might ask how a small change could cause the code in example 18.1 to crash. That would happen if our loop went to far. This is another of the big differences between Java and C. Java does bounds checking. That isn't just true of arrays, but of everything else. In C if you make an array with 5 elements and put things in the $6^{th}$ and $7^{th}$ elements

your code might crash, but more than likely it will just overwrite some other variable and keep running without complaint. Bugs created in this way are extremely hard to locate. That is why Java included bounds checking. If you ever try to use an array index less than 0, or greater than or equal to the number of elements, Java will throw an exception and crash that thread immediately.

What would happen if we changed example 18.1 to use an object type (like String) instead of a primitive type like an int? The answer is that it would be just like the change we have to make when we go from using a primitive type to using an object type. Arrays of object types in Java are arrays of references, not arrays of actual objects. All the references start off with a value of null. The first implication of this is that we need to initialize all the references with either a call to new or by assigning them the value of an existing object. The second implication is that arrays of objects are polymorphic. You can make an array of a given type and fill it with any subtype of that. This is very important for your game as you will have an array that holds Blocks, but you can put different subtypes of Block into it.

Just like in C, there are shortcuts for initializing arrays. Example 18.2 shows the declaration and initialization of several different arrays using both a normal method of storing values at indexes and the shortcut method of automatic initialization.

```
/**
 * Example 18.2
 * Just some code that declares and initializes a few arrays.
 */
public static void main(String[] args) {
	int[] aInt={1,2,3,4}; // shortcut syntax for ints.

	String[] aString=new String[3]; // make the array
	aString[0]="Now";
	aString[1]="fill";
	aString[2]="it";

	String[] aString2={"Now","fill","it"}; // same thing but shorter.

	Integer[] aInteger={1,2,3}; // only works because of autoboxing.

	Integer[] aInteger={new Integer(1),new Integer(2),new Integer(3)};
		// what the autoboxing is really doing.

	Block[] aBlock=new Block[2];
	aBlock[0]=new OpenBlock();
	aBlock[1]=new WallBlock();

	funcThatNeedsIntArray(new int[]{4,5,6});
}
```

The shortcut version of the declaration of a variable simply puts the values you want

inside of curly braces. Obviously, you aren't going to do this for large arrays. It is remarkably helpful for small arrays of primitive or String types. It is less useful for things where you have to instantiate all the elements as that requires significant typing by itself. The last two examples are somewhat more interesting. The example with the Block type shows that arrays are polymorphic. We can make one array and stick two types of Block objects in it. The last example is a lesser known feature of Java. It shows how you can use the shortened syntax with an array without actually declaring a variable to hold it.

Above I said that you make an array type by adding [] after the type you want an array of. Since an array type is itself a type, we can repeat this process to get types that are arrays of arrays, etc. This is how you get multidimensional arrays in Java. Simply give the type you want and put the proper number of brackets after it. So an int[][] is an array of arrays of ints. Something to keep in mind here is that arrays are object types and object types start off as null. Java have some shortcuts to help make your life a bit easier with this. Example 18.3 shows code for creating and initializing some 2-D arrays.

```
/**
 * Example 18.3
 * Displaying some use and initialization of two dimensional arrays.  Extension
 * to three or more dimensions is straightforward.
 */
public static void main(String[] args) {
	int[][] int2D=new int[5][6]; // simple syntax gives us 30 ints.

	int[][] int2Db=new int[5][]; // makes an array of references to int[].
	for(int i=0; i<int2Db.length; ++i) {
		int2Db[i]=new int[6]; // this makes all the actual ints.
	}

	String[][] names=new String[10][2]; // makes 20 null references to String

	String[][] names2=new String[10][]; // makes 10 references to String[]
	for(int i=0; i<names2.length; ++i) {
		names2[i]=new String[2]; // make the String references
		for(int j=0; j<names2[i].length; ++j) {
			names2[i][j]=""; // have the String references point to the
				// empty string.
		}
	}

	// this shows how the shortcut notation can be used with multidimensional
	// arrays.
	String[][] names3={{"John","Doe"},{"Jane","Doe"}};
}
```

The first and third examples show a simple syntax for making rectangular arrays. Those are arrays that have the same number of columns in each row. For primitives we again get the actual primitives and for objects we have null references. The second and fourth examples show a longer syntax where we allocate the second dimension of the array

separately. Note that multidimensional arrays created this way don't have to be rectangular. We could have each sub-array contain a different number of elements to make triangular arrays or ragged arrays or whatever we want.

Also note how we get the number of columns for the array of strings for our loop with the counter variable j. This syntax might seem odd, but if you think about it it will make perfect sense. The declaration String[][] names2 is actually making a reference variable called names2 that points to an array of arrays of Strings. When we do names2[i] for any index i, we are following the reference in names2 and pulling out the $i^{th}$ object in the array. If that object is itself an array, which it is in this case, then it will have a member named length to tell us the length of that array. The [] notation follows an array reference and gets something from the object just as the . notation does for any object. This is a significant thing for you to note when you are debugging. Perhaps the most common exception you will get when programming this semester is the NullPointerException. You get this when you tell Java to follow a reference and the reference is null. The exception stack trace will tell you what line it happened on. By realizing that [] and . follow references you can easily figure out what is null. When you get a NullPointerException you can look on the line it comes from and the null reference has to be to the left of either a . or a []. Most of the time it will be a ., but occasionally it could be the [].

## 19. Theads

Java has a number of advantages as a programming language and set of libraries that we will address in this class. Some, such as the ability to find bugs and the extended libraries for GUIs and graphics, you should instantly see the advantage of. One of the strengths of the Java language that you probably won't understand the advantages of early on is the ability to easily incorporate multithreading into your programs. All the programming that you have done so far has been single threaded. In fact, the very mental picture you have of how a program runs is likely single threaded. The image of the computer going from one statement to the next and executing them in the order specified by flow control statements is exactly what you should picture for a single threaded program. The idea of multithreading is that the computer is effectively doing 2 or more

statements in the program at the same time. Unlike multiprocessing, where two processes are running at the same time, the threads in a multithreaded program all have access to the same memory.

Letting a program do multiple things at once, and for those things to have access to the same memory at one time can lead to significant complexities. As you might have noticed, we teach an entire course on parallel programming. Half of the material in that course focuses on multithreading. We aren't going to get into all the details of multithreading, but we do want you to have exposure to it early on because the importance of multithreading in general programming is increasing every day.

The benefits of multithreading depend a lot on the machine that your program is running on. Machines with a single core (one processor with one core) only get the benefit of logical independence for the execution of threads. They can't really do two things at once so instead it executes one thread for a short period of time, then switches over and executes the other thread for a short period of time. The intervals are so short that to a human it appears the two are happing at the same time. Up through 2004 the vast majority of machines had a single core. Dual processor machines have been around a long time, but they typically cost significantly more and only workstation class machines have had them. Intel introduced “hyperthreading” before that time, but that doesn't provide true and full parallelism. In 2005 both Intel and AMD released dual core chips (IBM had done this even earlier, but not many people own POWER based systems). Machines based on these chips only have a single processor plugged into the motherboard, but that single processor actually has two fully independent cores etched into it and in most ways they behave exactly like a dual processor machine. These machine truly can execute two instructions at the same time. With the advent of dual core chips, the need for multithreading on home machines becomes much greater. For a single program to use the full power of a dual core machine, that program must have at least two threads. Otherwise, one of the cores will sit idle while the other core does all the work. This issue will only grow in significance over time as both Intel and AMD have or plan to release chips with 4 cores on them in 2006-2007. This trend for doubling the number of cores on chips roughly every two years is expected to continue at least until someone can figure out something better to do with the extra transistors allowed by

smaller etching methods. As it stands, Intel has talked about processors with 100 or more cores and the merger of AMD and ATI also points to a similar path for that chip maker.

Multithreading, and the concept of a thread, were built into Java from the original release of version 1.0. There is a java.lang.Thread class that encapsulates the idea of a thread of logic in a program. The fact that this class is in the java.lang package shows you how fundamental the makers of Java considered it to be. That's the same package that String, System, and the primitive wrapper classes are in. For this section we are just going to talk about the fundamentals of the Thread class and basic multithreading in Java. With Java 5.0 more threading support was added with the creation of the java.util.concurrent package. It contains a number of classes that help with common tasks where multithreading can help you. We will cover that package and how to use it later in the semester.

The process of making a separate thread of execution can be done with the Thread class. To make a thread that does what you want, you can extend the Thread class and implement a run method for it, or make a subclass of Runnable and pass that to the thread. We will work with the second method. Runnable is a very simple interface that has only a single method in it: void run(). When the start method is called on the thread object, the run method of the specified Runnable object is called in a separate thread. The original thread will continue running at the point right after the call of start, and the two will operate in parallel. The second thread will continue to execute until the run method in it terminates. You should not use the stop method of Thread to terminate a thread. That method has been deprecated. This code example shows a single statement in Java that can be used to start a thread running an arbitrary method.

```
/**
 * Example 19.1
 * This single statement creates a thread with a runnable object that will call
 * longRunningMethod() and begins that thread executing.  Notice that this
 * syntax doe not give you a reference to the thread if you wanted to have it
 * for some purpose later on.
 */
new Thread(new Runnable() {
      public void run() { longRunningMethod(); }
}).start();
```

While it is extremely easy in Java to create a new thread, more thought typically needs to go into how you are going to deal with that separate thread so that things happen the way you want. There are certain fundamental problems that one runs into when doing

multithreaded programming. These problems are the reason that most programs are not written to be multithreaded, and even why most current programmers do not know how to write multithreaded programs. The primary problem arises from the fact that different threads are able to access the same pieces of memory at the same time. A classic example of this is a bank account where processes are doing withdraws and deposits at the same time. An alternate example is to consider one thread searching through an array for an item while a second thread is sorting that same array. Obviously, the odds of the search thread getting the the correct answer are less than perfect.

```
/**
 * Example 19.2
 * This is a bank account class with withdraw and deposit methods.  I have
 * written those methods in a longer format than I might normally to make it
 * clear what steps they are doing.  Even if I wrote them in a single line of
 * code the computer would have to do what is written here.  I've put line
 * numbers on comments so I can refer to them in the text.
 */
public class BankAccount {
	/**
	 * This method will take funds out of the account. It returns the balance
	 * after the withdraw. Really this should throw an exception if there
	 * isn't enough money, but we will skip that here as it isn't needed for
	 * this example.
	 * @param value The amount to withdraw.
	 * @return The balance after the withdraw is done.
	 */
	public int withdraw(int value) {
		int ret=getBalance();
		ret-=value;
		setBalance(ret);
		return ret;
	}

	/**
	 * This method will put money into the account. It returns the balance
	 * after the deposit.
	 * @param value The amount to deposit.
	 * @return The balance after the deposit is done.
	 */
	public int deposit(int value) {
		int ret=getBalance();
		ret+=value;
		setBalance(ret);
		return ret;
	}

	/**
	 * This method returns the balance of the account.
	 * @return The balance in this account.
	 */
	public int getBalance() {
		return balance;
	}

	/**
	 * This method sets the balance of the account.
	 * @param bal The new balance in this account.
	 */
	public void setBalance(int bal) {
```

```
            balance=bal;
        }

        private int balance;
}
```

For the bank account example, consider a class with code in it as shown in example 19.2. Assume now that the application is multithreaded and that you happen to be making a withdrawl at the exact same time that your employer is making a deposit. This leads to what is called a race condition, and you don't want that. The problem is that both threads can call getBalance() at effectively the same time. One subtracts some money and the other adds some money. They both then call setBalance(), but one of them will call it slightly before the other. One of them will wind up writing to the memory after the other. There is no way around that. This means that one of the transactions will be completely lost. You might not mind that if the withdraw is lost, but if your paycheck for the month were lost it might cause you some problems. Either way, someone isn't going to be happy.

So the question is, how do you prevent this? The fundamental problem was that you can't have withdraw and deposit happening at the same time. For that matter, you can't have two withdraws or two deposits happening at the same time. You need some way to make sure that only one is happening at any given time. We do this with things called monitors. Fortunately for you, you don't have to deal directly with monitors. Java associates a monitor with every object and every class as needed. The way to tell Java that a method needs to be exclusive is to precede it with the **synchronized** keyword. When a method is synchronized, a call to the method with first check the monitor to see if it is locked. If it is not locked, that thread will lock it and proceed to execute the method. When it completes the method, the monitor will be unlocked. If you try to invoke a synchronized method and the monitor is locked, the thread will wait until the monitor has been unlocked before it actually executes the method. The effect of this is that only one synchronized method on a given object can be executing at any particular time.

So the solution to our problem with the bank account would simply be to have the deposit and withdraw methods be declared synchronized. You can also synchronize smaller blocks of code. The synchronized keyword can be put in a function followed by the object or class that should be locked in parentheses.

There is a significant risk to synchronization that you should be aware of. This is

the risk of deadlock. Deadlock happens when two threads try to take monitors on two objects in opposite order. So if thread A is synchronized on obj1 first, then on obj2 second while thread B is synchronized on obj2 first and obj1 second, it is possible that thread A will lock the monitor on obj1 and before it locks the monitor on obj2, thread B flips that lock. Now, when thread A gets to the code where it needs the monitor on obj2 it stops. Similarly, when thread B gets to the code needing the monitor on obj1 it also stops. At that point neither thread can proceed until the other one finishes so both are deadlocked. For this reason synchronization must be done judiciously.

Synchronization can't do everything you will want to do with threads. Its sole functionality is to prevent two threads from accessing sensitive pieces of code at the same time. There are times when you need to do more complex functions. For example, you might have a problem that you can break into several pieces and give each piece to a different thread and you need to do this repeatedly. However, you can't start the second time through until all the threads have finished the first one and some finalizing work has been done. The problem here is that you need some of the threads to pause their execution at the appropriate time and wait until they get a signal telling them that all the threads are done. The basic mechanism for doing this in Java is with the wait and notify methods in the Object class. The java.util.concurrent package adds other mechanisms to produce this same result that we will talk about later in the semester.

The wait method can be invoked on any object, but it needs to be invoked by a thread that holds the monitor for that object. Basically that means that you need to call it inside a synchronized method or block of code. Once you call wait, that thread will release the monitor and simply stop. It won't start executing again until another thread calls notify or notifyAll. The notifyAll method will wake up all the threads that are waiting on the monitor for that object. The plain notify method picks one waiting thread at random and wakes it up. There are two recommendations when using the notify/wait structure for handling threads. The simple one is to always use notifyAll and never use notify. Using plain notify can possibly wake up a thread that isn't ready to wake up and that doesn't notify anything when it goes back to into a wait state, leaving your code in limbo. Of course, it is possible that you didn't want to wake up everything. To deal with that, the second piece of advise is to always enclose your wait statements in a while loop

that checks a condition that will be false when the thread should actually wake up. Example 19.3 shows a method that you might use for the situation described above where several threads are going to work on a task and need to pause when they are done if the other threads aren't yet done.

```
/**
 * Example 19.3
 * This method could be called by various threads as they finish their tasks.
 * Threads go to sleep when they finish unless it is the last thread to
 * complete.  The last thread will notify the others to wake them up.  When
 * we originally start all the tasks, we set the boolean done to false and set
 * completed to 0.
 */
public synchronized void taskDone() {
       completed++;
       if(completed<totalThreads) {
              while(!done) {
                     wait();
              }
       } else {
              done=true;
              notifyAll();
       }
}
```

If you start a thread running and you want to wait until it has completed you can use the join method of the Thread object. This is a simpler case of what we have just talked about. If all you needed to do was wait until a number of different threads had completed, your original thread could spawn off the other threads and start them working, then call join on each thread in turn. This way the main thread would resume after all the other threads had completed.

Sometimes the tasks that you want to run in other threads might be more or less important than other things. For example, you might want to sort a large array, but the sorting doesn't need to happen immediately, you just want to start it now and it shouldn't interfere with the other things that are going on. In this situation you can alter the priority of a thread. The getPriority and setPriority methods of the Thread class allow you to tell Java if certain threads are more or less important than other ones. A lower priority thread will never execute if a higher priority one needs to be doing something.

There are a few static methods in the Thread class that can be helpful whether you are writing multithreaded programs or not. The sleep method allows you to tell Java that a thread should stop executing for a certain period of time. The yield method tells Java that this thread can stop executing right now, but it is willing to start back up the next time the scheduler comes to it. Doing a yield occasionally in a multithreaded program

can help make sure that certain threads don't hog the resources. Lastly, the dumpStack method will print out a stack trace at any point during program execution. This can be very helpful for programming debugging. It it the same output you get when an exception is thrown, only this method allows you to do it at any time you want. If a function is called from several locations, this method can be helpful for you to figure out where it is being called from when a certain error occurs.

Lastly, when you need something to happen at regular intervals Java has Timer classes that can help you. If your program is a graphical program you can use the javax.swing.Timer class. This actually doesn't create a new thread, it uses the standard Swing event processing thread to complete tasks. The java.util.Timer class does use a separate thread. It isn't as safe for graphical displays, but it will give you much better performance on multiprocessor machines.

## 20. GUIs (Graphical User Interfaces)

In the previous section we discussed threading as an advantage of the Java platform. The way that Java supports threads is much simpler than the support in C/C++ and it has been there since the language was originally created. This same type of statement can also be made of GUIs. You are inevitably most familiar with working with computers through GUIs. Command lines interfaces are far more rare these days than they were in the early days of programming. You can do GUI programming in most languages. However, most languages don't have GUIs as part of their standard libraries, and as a result, the GUIs that you create typically aren't very portable. Java has included support for GUIs in the standard library since Java 1.0. At that time GUIs were written using the java.awt library. AWT standards for “Abstract Windowing Toolkit”. It is basically a thin layer of code that sits on top of whatever operating system you happen to be running on. So when you create GUI elements in AWT it actually passes that straight down to the OS, which creates them. This has some advantages for speed, but it lacks flexibility and portability. Because of the way AWT works, it only includes GUI elements that are present in all the OSes or windowing environments and you don't have much of any control over how those elements appear.

For these reasons, the Swing library was added as an extension with Java 1.1.

Swing is “pure Java”. The actual drawing of the GUI components is written in Java code so Java has complete control over how everything is drawn. The downside of this approach historically has been speed and when Swing first came out, many developers avoided it simply because the performance was quite poor. Over time that has been corrected and Swing now runs at virtually the same speed as AWT. In addition to being portable, the pure Java nature of Swing has also allowed the library to include quite a few components that couldn't be in AWT because they are general across platforms. These include simple components like sliders and spinners as well as complex components like tables and trees.

As it turns out, many other things have changed since Java 1.0 and the original AWT that we won't get into. For the rest of this section we will focus on how you should be programming in Swing. You can go to look at the tutorial online for more examples of how to do things. This will be more of a ground up description.

Everything in both AWT and Swing is based on the Component. For Swing, all things are based on JComponent, which itself inherits from Component. Anything that you can put into a GUI other than the top level windows will be a JComponent. This includes components that can hold other components. These are called containers. We will use the JPanel class for our containers. That class will also be helpful later on when we do custom graphics. Before we get into the use of containers, lets briefly describe some of the components that are available in Swing. The following list includes most of the basic component classes in Swing. Notice that classes in the Swing library often start with J to differentiate them from their AWT counterparts. The classes in alphabetical order are JButton, JCheckBox, JComboBox, JEditorPane, JFormattedTextField, JLabel, JList, JProgressBar, JRadioButton, JScrollBar, JSlider, JTable, JTextArea, JTextField, and JTree. Some of these, like the button and check box are rather standard. Others, like the editor pane, table, and tree are rather advanced. The existence of the advanced components in the standard library can potentially save you a lot of time as a programmer.

Every GUI needs to have some top level object in order for things to be visible. In Swing there are three possible top level objects: JFrame, JDialog, and JApplet. The last is only used if you are writing an applet for a web browser. A frame is a standard

window and that is what we will use most of the time. The dialog is used for dialog boxes and has the ability to block the user from accessing other windows while it is up (that is to say it can be modal). Our first example program for this section will create a window and put a button in that window.

```
/**
 * Example 20.1
 * This creates a frame and puts a button in it.  It then displays the frame.
 */
public class GUIDemo {
    public static void main(String[] args) {
        buildGUI();
    }

    private static void buildGUI() {
        JFrame frame=new JFrame("Example 20.1");
        JButton button=new JButton("My Button");
        frame.add(button);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.pack();
        frame.setVisible(true);
    }
}
```

We start off my creating a frame and giving it a title. We then make a button that has the text “My Button” on it. We add that button to the frame and tell the frame that when it closes, the application should terminate. You only do that for the main window in an application. You'd hate for your application to terminate when some secondary window is closed. The next line of code calls the pack method on the frame. This tells the frame to lay things out and determine a minimum size for everything to fit nicely. Last we set the frame to be visible so it shows up on the screen. The result of running this program is shown in the figure below. At this point, the button doesn't do anything


*Figure 4: This is the window that pops up when you run example 20.1.*

and even more restricting, we don't know how to add more than one button to our frame.

To have more than one component inside of our frame we need to learn something about containers in Java. As was mentioned above, a container is a component that can hold other components. Our frame is a container. Technically, when we called frame.add() we were really doing frame.getContentPane().add(), but Java 5.0 does the forwarding for you. The question with putting multiple things inside of a container is,

where do they all go? While you can specify exactly what pixel coordinates components go at, this practice is strongly frowned upon in Java because it produces rigid GUIs that aren't very portable. Instead, you should use layout managers to control where components go inside of a container. Swing and AWT come with several different layout managers. They all inherit from the java.awt.LayoutManager interface. The more standard layout managers are the following: BorderLayout, BoxLayout, CardLayout, FlowLayout, GridBagLayout, GridLayout, and SpringLayout. There are other layouts that you won't use directly, but that are employed by some of the advanced containers we'll talk about a bit later. You can read about all of these layouts and how they work in the API. We will talk briefly about a few of them so that you can see how they are used.

You can specify a layout manager for a container with the setLayout method. In some cases, like with JPanel, you can also specify the layout as an argument to the constructor. The simplest layouts are FlowLayout and GridLayout. FlowLayout places components one after another in line then wraps around like a text editor would do if there are more things than fit on a single line. The GridLayout gets a fixed number of rows and columns and divides the containers into a regular grid where all the cells are the same size.

Even with these two simple layouts we have a bit of power in what we create because we have the ability to nest containers inside of one another. So we could have a grid with flow layouts inside of it or a flow layout with grids inside of it. Example 20.2 shows a method that does the former. We have a grid with two rows and we add into it two panels that are using flow layouts.

```
/**
 * Example 20.2
 * This demonstrates the use of layouts and nesting containers.  It has a
 * GridLayout in the frame and uses FlowLayouts in panels that go in the two
 * grid cells.
 */
private static void buildGUI2() {
    JFrame frame=new JFrame("Example 20.2");
    frame.setLayout(new GridLayout(2,1));
    JPanel panel=new JPanel(new FlowLayout());
    panel.add(new JLabel("Top Button"));
    JButton button=new JButton("Button 1");
    panel.add(button);
    frame.add(panel);
    panel=new JPanel(new FlowLayout());
    panel.add(new JLabel("Bottom Text Field"));
    JTextField textField=new JTextField("Starting text");
    panel.add(textField);
    frame.add(panel);
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

```
    frame.pack();
    frame.setVisible(true);
}
```

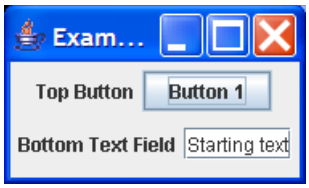


*Figure 5: This is the frame produced by the code in example 20.2.*

If we replace the call to buildGUI in example 20.1 with buildGUI2 we get the image shown in this figure. You can see how the GridLayout is allowing us to stack elements vertically while the FlowLayouts place objects side by side. In the FlowLayout, each element gets as much space as it "wants" to take. Every component has methods it can use to tell layout managers what the minimum, maximum, and preferred sizes are for that component. You can tell that for the standard components, those sizes are determined by the things like how long the text string in a label is.

The next layout manager up in complexity is the BorderLayout. Despite the simplicity of this layout it is remarkably useful when nested with other layouts. A BorderLayout defines 5 regions that you can put components in. They are north, south, east, west, and center. Each can hold one component, which might be a container. The components on the north and south take up the full width and are given as much space as they want to have vertically. East and west go from the bottom of north to the top of south and each take up as much room as they want horizontally. Whatever is left after those 4 are done becomes space for center. If any side region doesn't have components in it, then it doesn't take up any space and isn't drawn.

Example 20.3 shows a usage of a BorderLayout with both GridLayout and FlowLayout being used in nested components. In this example, components have been placed in grids in east, west, and center. A FlowLayout is used in south and nothing is placed in north to show you how nothing is shown in that case. The side grids have labels and buttons in them. The central grid has text fields. Instead of using pack here we specify a size for the window as the text fields start off empty which makes their preferred size quite small.

```
/**
 * Example 20.3
 * This demonstrates the use of BorderLayout and nesting containers.
```

```
 */
private static void buildGUI3() {
    JFrame frame=new JFrame("Example 20.3");
    frame.setLayout(new BorderLayout());
    String[] labelText={"Name","Address","City, State"};
    JPanel panel1=new JPanel(new GridLayout(labelText.length,1));
    JPanel panel2=new JPanel(new GridLayout(labelText.length,1));
    JPanel panel3=new JPanel(new GridLayout(labelText.length,1));
    for(int i=0; i<labelText.length; ++i) {
        panel1.add(new JLabel(labelText[i]));
        JTextField field=new JTextField();
        panel2.add(field);
        JButton button=new JButton("Update");
        panel3.add(button);
    }
    frame.add(panel1,BorderLayout.WEST);
    frame.add(panel2,BorderLayout.CENTER);
    frame.add(panel3,BorderLayout.EAST);
    JPanel southPanel=new JPanel(new FlowLayout());
    JButton button=new JButton("Done");
    southPanel.add(button);
    button=new JButton("Cancel");
    southPanel.add(button);
    frame.add(southPanel,BorderLayout.SOUTH);
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.setSize(400,170);
    frame.setVisible(true);
}
```

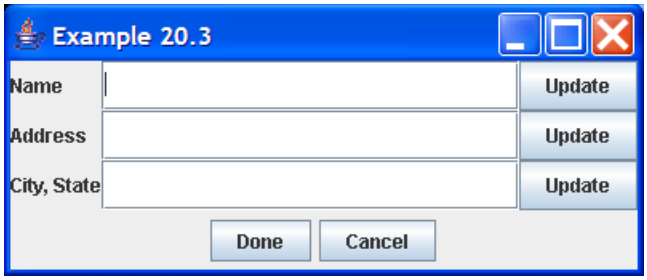


*Figure 6: This is the window produced by example 20.3 combining border, grid, and flow layouts.*

You should note one very significant difference between the usage of BorderLayout and the other layouts. When you add a component into a container with a BorderLayout, you have to specify a second argument that tells it what region to put it in. You should use the constants defined in BorderLayout for this. They are just strings, but it is safer to use them than to type in string constants yourself.

Looking at the other layout managers, CardLayout provides functionality for swapping one component at a time to the top. We will get this with a JTabbedPane later on. GridBagLayout is an extremely flexible and powerful layout manager that is beyond the scope of what we will discuss here. BoxLayout provides something similar to a grid with a single row or column. The different is that each box in it can be a different size. If you want to use a BoxLayout you will probably find it easiest to simply use the Box container type which defaults to it. SpringLayout is another layout manager that

dynamically positions components, but once again it is more complex than what we will go into here. The Java API contains descriptions of these classes as well as links to tutorials for many of them.

Swing provides some container classes with specialized layout managers that you can use for doing certain tasks that would be hard to accomplish with standard layout managers. These include JDesktopPane, JLayeredPane, JScrollPane, JSplitPane, and JTabbedPane. Again, you can find full details on these in the API and often the API includes links to tutorials on how to use them. We will briefly discuss the last three as they are generally useful for building functional GUIs.

The JScrollPane holds a single component inside of it. When that component is bigger than the area the scroll pane can display, scroll bars are automatically added to the sides as needed so that users can scroll to see the parts that are hidden. The JSplitPane holds two components that are either drawn side by side, or stacked one above the other. With the JSplitPane, the dividing line between the two components can be moved by the user to partition the display area as they see fit. Lastly, the JTabbedPane provides an easy way for the user to flip between different components by clicking on tabs. This type of behavior is commonly seen in dialog boxes for the settings/options in Windows programs.

Another feature that Swing provides that you can use to make your GUIs more visually appealing is the ability to add borders around components. Typically you will put a border on a panel to offset the elements that are inside of it and distinguish them from other elements. Every JComponent has a setBorder method that you can call to set a border that will be drawn around that component. The best way to get a border is with the javax.swing.BorderFactory class. This class has a large number of static methods than can be used to produce many different types of borders, with and without text labels. Since we aren't focusing too much on making our GUIs pretty for this class we won't go into further detail on borders.

Another aspect that adds significantly to GUIs is the ability to add menus to frames for users to select common options from. Menus often look a lot nicer for certain options than a collection of buttons would. In Swing, a menu across a window is actually a JMenuBar. You can set one of these per frame or dialog with the setJMenuBar method.

You can add JMenu objects into the menu bar. Those objects represent the actual menus that you can pull down. You can add either JMenuItems or other JMenu objects to a JMenu. The former become individual selections while the latter will give you submenus. If you want menu items that can be selected like check boxes or radio buttons, there are also classes for JCheckboxMenuItem and JRadioButtonMenuItem. For more information on these and other information on menus see the API. Swing also has support for popup menus though we won't be discussing them here. We will deal a bit more with menus below.

**20.1 Event Handling**

Given what has been discussed so far, you now have the knowledge to put together basic GUIs in Swing. The only problem is that they don't do anything. We put in buttons, but nothing happens when we click them. We put in text fields, but we have no idea how to do anything with the text that a user might type into them. In order to truly have a useful GUI, we have to be able to deal with user input on that GUI.

Input in a GUI is very different from input in a command line program. In C, you dealt with the scanf function for getting input from a user. This function would read in whatever the user had typed at the console. If nothing had been typed, it would wait until the user did type something in. That entire style of input doesn't work with a GUI. There are numerous reasons for that. In a GUI the user can be giving you input from the keyboard of from the mouse. A single GUI might have 100 things that the user can click on or type into and each one might do something different. Lastly, in a GUI the graphical display might be actively doing something while the user is typing or clicking.

All these factors together point to a need for a very different way of handling user input with a GUI. It can't be a single, centralized, blocking method like scanf. Instead, each component in a GUI needs to be able to specify code that will happen at certain times and that code needs to not interfere with the program if the user isn't providing any input. The way this has been done in both AWT and Swing since Java 1.1 is with Listeners. Each and every component can have a number of listeners of various types registered with it, so that when different events occur, the proper code in the listener is called and executed. AWT provides a number of different listener interfaces. It is up to

you to provide an implementation. The fact that listener classes are typically small and they come up so frequently in GUI programming was actually a major motivation for the addition of anonymous inner classes in Java. We will be using those extensively in the examples that follow. For that reason, if you aren't comfortable with these, you should consider going back and reading the section on inner classes again.

There are quite a few different Listener interfaces defined in the java.awt.event and javax.swing.event packages. You can go look in the API to see the complete list. We will demonstrate the use of events with ActionListener and FocusListener. We start with the simple clicking of a button. This is an example of a use for an ActionListener. Components that have simple styles of interaction can use the ActionListener. It has a single method in it called actionPerformed which takes a single argument of type ActionEvent. In example 20.3 after making the button with the label "Done", we could have put in the following.

```
/**
 * Example 20.4
 * Writing an ActionListener for a button.
 */
button.addActionListener(new ActionListener() {
	public void actionPerformed(ActionEvent e) {
		doProcessingOfStuff();
		System.exit(0);
	}
});
```

This would make it so that when the user clicks on the "Done" button the program will call doProcessingOfStuff and then exit. The doProcessingOfStuff method would be a different private method inside of our class. For the "Cancel" button we would probably use the same code only leave out the line for doing the processing.

Notice that we are creating a anonymous inner class here. We specify the supertype, in this case ActionListener, then provide the body of the class with an implementation of any methods that are needed. Because ActionListener has only one method, that is all we have to write. The call to an outside function is done not only because I don't have real code to put in there for this example, but also to keep the code short. Remember that anonymous inner classes are typically not very long and making them long can cause code to be confusing. A few lines in the body of the method is ideal. Often this means you will put in a call to an outside function.

In the example GUI code above, we included not only buttons, but also text fields. It turns out that a JTextField can also fire an ActionEvent that should be noticed by an

ActionListener. This happens when you press Enter while you are editing the text field. Often you want it so that the text in a field is processed when the field loses focus in addition to when the user presses Enter. For this you need a FocusListener. The FocusListener is more complex than the ActionListener in that it has two methods: focusGained and focusLost. We only care about when the focus is lost, not when it is gained. We could certainly implement FocusListener and make a blank implementation of focusGained, but the Java libraries provide us with an alternative. All of the listener interfaces that have more than one method have a corresponding adapter class. This class implements all the methods of the interface with blank method bodies. So for processing a text field, we might have code like the following.

```
/**
 * Example 20.5
 * Writing listeners to handle events on a JTextField.
 */
final JTextField field=new JTextField();
field.addActionListener(new ActionListener() {
	public void actionPerformed(ActionEvent e) {
		processString(field.getText());
	}
});
field.addFocusListener(new FocusAdapter() {
	public void focusLost(FocusEvent e) {
		processString(field.getText());
	}
});
```

Here we have made it so the same processing happens whether we press enter or the field loses focus. Note that the field variable is declared to be final. We have to do this because it is used in the body of the inner class. You should recall that inner classes have access the member data as well as final local variables. If you leave the final off, this code would not compile. In this case, I've assumed that the processing only needs the string from the text field. Sometimes, the function might need to alter the text field depending on what is typed in. In that case, we could either pass field to the function, or take the declaration of field out of this function and make it a member variable.

To close this out, we will write a GUI that is a bit more functional. It will have a layout similar to example 20.3, but will include listeners to handle user input and other stuff. Some of the code will be left incomplete as it would probably be best to do from a file. We'll assume for our purposes here that this frame is embedded in some larger application. That application constructs an object of type PersonGUI and passes it an array of Person objects. The point of this frame is that it lets you search that array by first

name, last name, and/or zip code, then gives a list of the hits and lets the user edit fields in the selected Person object.

```
/**
 * Example 20.6
 * This is a larger example of a GUI that has listeners attached to the
 * components to act on user input.
 */
public class PersonGUI {
    public PersonGUI(Person[] p) {
        people=p;

        frame=new JFrame("Edit People");
        frame.setLayout(new BorderLayout());
        JPanel northPanel=new JPanel(new GridLayout(4,1));
        JPanel panel=new JPanel(new BorderLayout());
        panel.add(new JLabel("First Name"),BorderLayout.WEST);
        final JTextField firstSearch=new JTextField();
        panel.add(firstSearch,BorderLayout.CENTER);
        northPanel.add(panel);
        panel=new JPanel(new BorderLayout());
        panel.add(new JLabel("First Name"),BorderLayout.WEST);
        final JTextField lastSearch=new JTextField();
        panel.add(lastSearch,BorderLayout.CENTER);
        northPanel.add(panel);
        panel=new JPanel(new BorderLayout());
        panel.add(new JLabel("First Name"),BorderLayout.WEST);
        final JTextField zipSearch=new JTextField();
        panel.add(zipSearch,BorderLayout.CENTER);
        northPanel.add(panel);
        panel=new JPanel(new GridLayout(1,2));
        JButton button=new JButton("Search");
        button.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                doSearch(firstSearch.getText(),lastSearch.getText(),zipSearch.g
etText());
            }
        });
        panel.add(button);
        button=new JButton("Clear");
        button.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                firstSearch.setText("");
                lastSearch.setText("");
                zipSearch.setText("");
            }
        });
        panel.add(button);
        northPanel.add(panel);
        frame.add(northPanel,BorderLayout.NORTH);

        hitList=new JList();
        hitList.addListSelectionListener(new ListSelectionListener() {
            public void valueChanged(ListSelectionEvent e) {
                currentPerson=(Person)hitList.getSelectedValue();
                setDisplay();
            }
        });
        frame.add(new JScrollPane(hitList),BorderLayout.CENTER);

        JPanel southPanel=new JPanel(new GridLayout(3,1));
        nameField=new JTextField();
        nameField.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) { updateName(); }
        });
        nameField.addFocusListener(new FocusAdapter() {
```

```
        public void focusLost(FocusEvent e) { updateName(); }
    });
    southPanel.add(nameField);
    addressField=new JTextField();
    addressField.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) { updateAddress(); }
    });
    addressField.addFocusListener(new FocusAdapter() {
        public void focusLost(FocusEvent e) { updateAddress(); }
    });
    southPanel.add(addressField);
    panel=new JPanel(new GridLayout(1,2));
    cityField=new JTextField();
    cityField.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) { updateCity(); }
    });
    cityField.addFocusListener(new FocusAdapter() {
        public void focusLost(FocusEvent e) { updateCity(); }
    });
    panel.add(cityField);
    JPanel innerPanel=new JPanel(new BorderLayout());
    stateBox=new JComboBox(Person.stateCode);
    stateBox.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) { updateState(); }
    });
    innerPanel.add(stateBox,BorderLayout.WEST);
    zipField=new JTextField();
    zipField.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) { updateZip(); }
    });
    zipField.addFocusListener(new FocusAdapter() {
        public void focusLost(FocusEvent e) { updateZip(); }
    });
    innerPanel.add(zipField,BorderLayout.CENTER);
    panel.add(innerPanel);
    southPanel.add(panel);
    frame.add(southPanel,BorderLayout.SOUTH);

    frame.setSize(400,400);
    frame.setVisible(true);
}

private void doSearch(String first,String last,String zip) {
    List<Person> hits=new LinkedList<Person>();
    for(int i=0; i<people.length; ++i) {
        if((first.equals("") || first.equals(people[i].getFirst())) &&
                (last.equals("") || last.equals(people[i].getLast())) &&
                (zip.equals("") || zip.equals(people[i].getZip()))) {
            hits.add(people[i]);
        }
    }
    hitList.setListData(hits.toArray());
}

private void setDisplay() {
    if(currentPerson==null) {
        nameField.setText("");
        addressField.setText("");
        cityField.setText("");
        zipField.setText("");
    } else {
        nameField.setText(currentPerson.toString());
        addressField.setText(currentPerson.getAddress());
        cityField.setText(currentPerson.getCity());
        zipField.setText(currentPerson.getZip());
        stateBox.setSelectedItem(currentPerson.getState());
    }
```

```
    }

    private void updateName() {
        if(currentPerson==null) {
            nameField.setText("");
        } else {
            String[] parts=nameField.getText().split(" +");
            currentPerson.setFirst(parts[0]);
            currentPerson.setLast(parts[parts.length-1]);
        }
    }

    private void updateAddress() {
        if(currentPerson==null) {
            addressField.setText("");
        } else {
            currentPerson.setAddress(addressField.getText());
        }
    }

    private void updateCity() {
        if(currentPerson==null) {
            cityField.setText("");
        } else {
            currentPerson.setCity(cityField.getText());
        }

    }

    private void updateState() {
        if(currentPerson!=null) {
            currentPerson.setState(stateBox.getSelectedItem().toString());
        }
    }

    private void updateZip() {
        if(currentPerson==null) {
            zipField.setText("");
        } else {
            currentPerson.setZip(zipField.getText());
        }

    }

    private Person[] people;
    private JFrame frame;
    private JList hitList;
    private JTextField nameField;
    private JTextField addressField;
    private JTextField cityField;
    private JComboBox stateBox;
    private JTextField zipField;

    private Person currentPerson;
}
```

*Figure 7: Output of example 20.6 for a random person.*

The toString method of Person has been overloaded to return the first and last names separated by a space. All the other methods should be fairly self-explanatory. Notice how the GUI uses nested containers to provide the different elements in places that are reasonable for input.

## 21. Custom Drawing and Java2D

We have talked about how to set up GUIs, but what do you do if you want to show something or do something that isn't a standard GUI component? In that situation you have to draw it yourself. We are going to deal with that situation in this section.

There are several steps to doing custom drawing in a Java application. First, you need to create your own class that is a JComponent. We will be making subclasses of JPanel. This component is what we will add into the GUI and it will actually display the custom drawing that we want. To get it to do this, we override the paintComponent method of JComponent. This method is a protected method that takes a single argument of the type java.awt.Graphics. The Graphics object is an object that encapsulates drawing capabilities in Java. Anything that you draw to the Graphics object will appear on the screen. An extremely simple example of this appears in example 21.1.

```
/**
 * Example 21.1
 * This is a simple example showing a component that can be added to a GUI that
```

```
 * will appear as just a black box.
 */
public class BlackBox extends JPanel {
    protected void paintComponent(Graphics g) {
        g.setColor(Color.black);
        g.fillRect(0,0,getWidth(),getHeight());
    }
}
```

This example shows extremely simple usage of the Graphics object. All we do is set the color that it is drawing with and then fill in a rectangle that is as large as the component is. That will make the entire component appear as a black box.

This method of setting the color and using a method like fillRect to fill in a rectangle is no longer the ideal method of doing custom drawing in Java. Instead, we should use the capabilities in Java2D. For this reason, we will not go into significant detail on how to draw with a Graphics object and instead we will focus on how to use the Graphics2D class. If you ever need to write code for a platform that doesn't have full Java and Java2D, you can look into how to work with the basic Graphics object yourself. As it happens, the Graphics object, g, will always be an instance of Graphics2D, which is a subclass of Graphics. The Graphics2D class is the heart of the Java2D API. Java2D gives us significantly more power than the older drawing routines. We will look a bit at what those capabilities are, but first let's see what the example above would have looked like using Java2D.

```
/**
 * Example 21.2
 * This is a simple example showing a component that can be added to a GUI that
 * will appear as just a black box.  It uses Java2D.
 */
public class BlackBox extends JPanel {
    protected void paintComponent(Graphics gr) {
        Graphics2D g=(Graphics2D)gr;
        g.setPaint(Color.black);
        g.fill(new Rectangle2D.Double(0,0,getWidth(),getHeight()));
    }
}
```

It should be noted that the code in example 21.1 is perfectly valid Java code and will compile and run without problems. The Java2D capabilities are an extension, not obligatory. The code itself using Java2D looks remarkably similar to the original version. It has a few differences that are quite significant. The first line takes the argument that was passed in as a Graphics object and casts it to a Graphics2D object. This operation is safe any time you are using Swing. Instead of calling setColor we call setPaint. The setColor method still works, but the setPaint method is more general. Color is a subtype of Paint so we can pass in a color just like before, but there are other options for Paint

that we will look at later. Also, instead of calling fillRect we call the more general method, fill, and pass it a rectangle. The fill method takes objects of type Shape. Shape is a very general type that can represent not only simple things like rectangles and ovals, but more complex shapes as complex as fonts for strings. As we will see, Java2D does almost everything through the process of filling in these shapes with various paints.

The setPaint method is one of several set methods that determine how the Graphics2D object will actually draw the things that you ask it to draw. The other methods are setBackground, setClip, setColor, setComposite, setFont, setPaintMode, setRenderingHint, setStroke, setTransform, and setXORMode. We will only talk about a few of these. You have already seen setColor and setPaint in use. They basically tell the Graphics2D object what color(s) to draw with when it draws. There are three main subclasses of Paint that you are likely to use. The simplest one is Color, which we have already seen. This class lets you combine red, green, and blue components to make whatever colors you want.

Another subclass of Paint is the GradientPaint. This class keeps two colors and two points in 2-D space. It causes the color to change linearly from one color to the other as you move from one point to the other. You can also tell this class if the fill should be cyclic or not. To see how this matters, assume the two points were (40,0) and (50,0) and the colors were red and blue. So the color would vary moving across the x-direction and it would be red when x=40 and blue when x=50. If it is set to be non-cyclic, then everything below 40 will be red as well and everything above 50 will be blue. If it is set to be cyclic then the color will go from red to blue as you move from 40 down to 30 then back to red as you go to 20. Basically, the colors cycle.

The third Paint subclass in the standard libraries is TexturePaint. When you set a graphics object to use a texture paint, everything that is filled in will be done with the image that you provide when you make the TexturePaint object. You also pass a rectangle, and the image will be mapped into that rectangle. The image is repeated in rectangles tiled around the one you specify.

So let's put these things together and write an application that will bring up a frame and draw our panel in it. We'll do a drawing that uses different paint selections for filling some different shapes.

```
/**
```

```
 * Example 21.3
 * This is an example of using different paints to fill in different shapes
 * it has been put into a complete little program that brings up a frame and
 * shows our panel it in.
 */
public class CustomDrawing extends JComponent {
    public static void main(String[] args) {
        JFrame frame=new JFrame("My Drawing");
        frame.add(new CustomDrawing());
        frame.setSize(500,500);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }

    protected void paintComponent(Graphics g) {
        Graphics2D g2=(Graphics2D)g;
        g2.setBackground(Color.white);
        g2.clearRect(0,0,getWidth(),getHeight());
        g2.setPaint(Color.red);
        g2.fill(new Rectangle2D.Double(20,20,100,100));
        g2.setPaint(new GradientPaint(40,40,Color.blue,60,50,Color.green,true));
        g2.fill(new Ellipse2D.Double(150,20,100,100));
        if(img==null) {
            try {
                img=ImageIO.read(new File("../../scene.png"));
            } catch(IOException e) {
                e.printStackTrace();
            }
        }
        g2.setPaint(new TexturePaint(img,new
Rectangle2D.Double(280,20,100,100)));
        g2.fill(new RoundRectangle2D.Double(280,20,100,100,10,10));
    }

    private BufferedImage img;
    private static final long serialVersionUID = 4578330655011242999L;
}
```

So when you call fill with a shape it uses the current paint style to fill in that shape. What happens when you call draw? By default you will see a thin line that draws the outline of the shape and it will be done with the current fill style. However, in reality, it is doing a Stroke around the edge of the shape and using that to build a new shape that is then filled. You can alter the Stroke with the setStroke method on Graphics2D. The standard libraries includes only one implementation of Stroke, but it is rather full featured so you aren't likely to need anything more. That implementation is BasicStroke and the constructors tell you what you can do with it. If you look at the API you will see a number of different constructors each with more arguments than the last. We'll only look at the last, though shorter versions can be used if you don't want to specify everything.

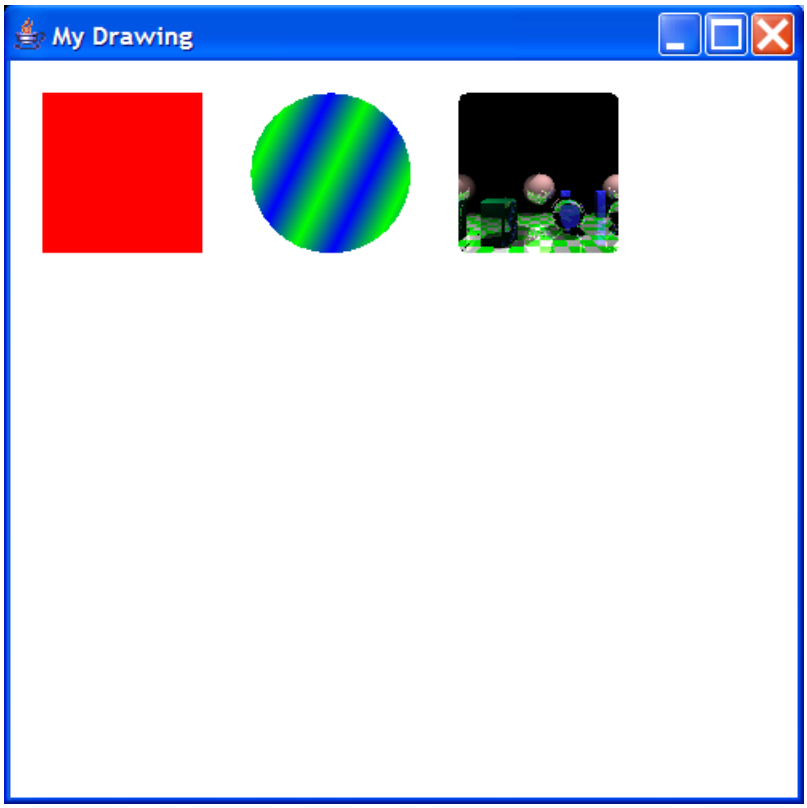


*Figure 8: This shows the window that is brought up by example code 21.3. The three different shapes are filled with a standard Color, a GradientPaint, and TexturePaint.*

The first argument is a width. This tells the stroke how wide to make the line. By default it is one. That is followed by a cap setting, which tells the stroke how to end lines. The options are CAP_BUTT, CAP_SQUARE, and CAP_ROUND. The names are fairly self-explanatory. The next argument is a join setting. This tells the stroke what to do about the angle joints between lines. If the width is very large this can be significant. The options here are JOIN_BEVEL, JOIN_MITER, and JOIN_ROUND. There is also a miterLimit argument that is only used if you set join to JOIN_MITER. The bevel join will put a straight line between the outer corners you would get by making each segment a block of the proper width. The round join does what the name implies. The miter join extends the outer lines on the two blocks until they meet making a sharp point. If that would go further than the miter limit then the lines bend so they meet closer in.

The last arguments to the longest BasicStroke constructor are an array of floats and a single float that can be used to form dashed lines. The array specifies how much should be drawn, followed by how much should be left out in a repeating pattern. The last argument says how far to draw the line before the dashing starts. So a simple dashed

line could be made with an array something like {3,3}. That would do three pixels on followed by three off. A dash-dot line could be done with {3,2,1,2}. This would have 3 on, followed by two off, followed by one on, followed by two off then repeating that.

```
/**
 * Example 21.4
 * This code adds to the code for example 21.3 to display the use of
 * BasicStroke.  To keep the listing smaller we are only showing the
 * paintComponent method.
 */
protected void paintComponent(Graphics g) {
    Graphics2D g2=(Graphics2D)g;
    g2.setBackground(Color.white);
    g2.clearRect(0,0,getWidth(),getHeight());
    g2.setPaint(Color.red);
    java.awt.Shape s=new Rectangle2D.Double(20,20,100,100);
    g2.fill(s);
    g2.setPaint(Color.cyan);
    g2.setStroke(new BasicStroke(10));
    g2.draw(s);
    g2.setPaint(new GradientPaint(40,40,Color.blue,60,50,Color.green,true));
    s=new Ellipse2D.Double(150,20,100,100);
    g2.fill(s);
    g2.setPaint(new GradientPaint(20,20,Color.red,10,30,Color.orange,true));
    g2.setStroke(new BasicStroke(10,BasicStroke.CAP_BUTT,BasicStroke.JOIN_BEVEL
            ,10,new float[]{20,15,10,15},0));
    g2.draw(s);
    if(img==null) {
        try {
            img=ImageIO.read(new File("../../scene.png"));
        } catch(IOException e) {
            e.printStackTrace();
        }
    }
    g2.setPaint(new TexturePaint(img,new Rectangle2D.Double(280,20,100,100)));
    g2.fill(new RoundRectangle2D.Double(280,20,100,100,10,10));

    g2.setPaint(Color.black);
    GeneralPath gp=new GeneralPath();
    gp.moveTo(20,200);
    gp.lineTo(100,200);
    gp.lineTo(20,250);
    g2.setStroke(new
BasicStroke(10,BasicStroke.CAP_BUTT,BasicStroke.JOIN_BEVEL,10));
    g2.draw(gp);
    gp=new GeneralPath();
    gp.moveTo(120,200);
    gp.lineTo(200,200);
    gp.lineTo(120,250);
    g2.setStroke(new
BasicStroke(10,BasicStroke.CAP_ROUND,BasicStroke.JOIN_ROUND,10));
    g2.draw(gp);
    gp=new GeneralPath();
    gp.moveTo(220,200);
    gp.lineTo(300,200);
    gp.lineTo(220,250);
    g2.setStroke(new
BasicStroke(10,BasicStroke.CAP_SQUARE,BasicStroke.JOIN_MITER,100));
    g2.draw(gp);
}
```

The next setting we will look at is for drawing text to the Graphics2D object. You can use the setFont method to specify what font you want your text to be drawn in. The

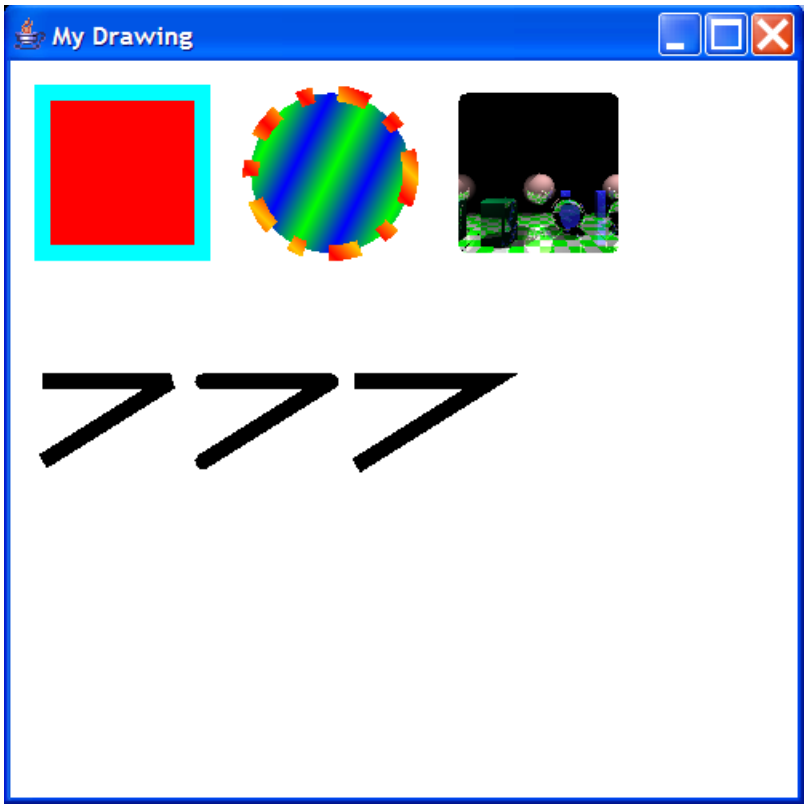


*Figure 9: This shows the frame that pops up when we run example 21.4. Notice the different uses of BasicStroke and the demonstration of joins and caps.*

Font class will let you build an object with a specified font name, a point size, and a style. We won't get too much into the details of how to deal with fonts or place them properly, but some introduction is needed. The drawString method will use the specified font and convert the glyphs of the letters for the font into a shape. It then does a fill on that shape. So the paint setting that you have will be used for the font as well. The position that is passed into the drawString method will be the beginning of the first character horizontally and it will be where the line on ruled paper would be vertically. So some of the font will go below that. If you want to know the exact size of the string you will be drawing you need to use the getLineMetrics and/or getStringBounds in the Font class.

```
/**
 * Example 21.5
 * These are two lines that can be added to to end of 21.4 to display a string.
 */
g2.setFont(new Font("Serif",Font.ITALIC,40));
g2.drawString("This is a string.",20,300);
```

Another setting that you can use when drawing to a Graphics object is the ability to set the clipping region with setClip. The clipping region tells the graphics object where it can draw. If you try to draw outside of the clipping region, the stuff outside will

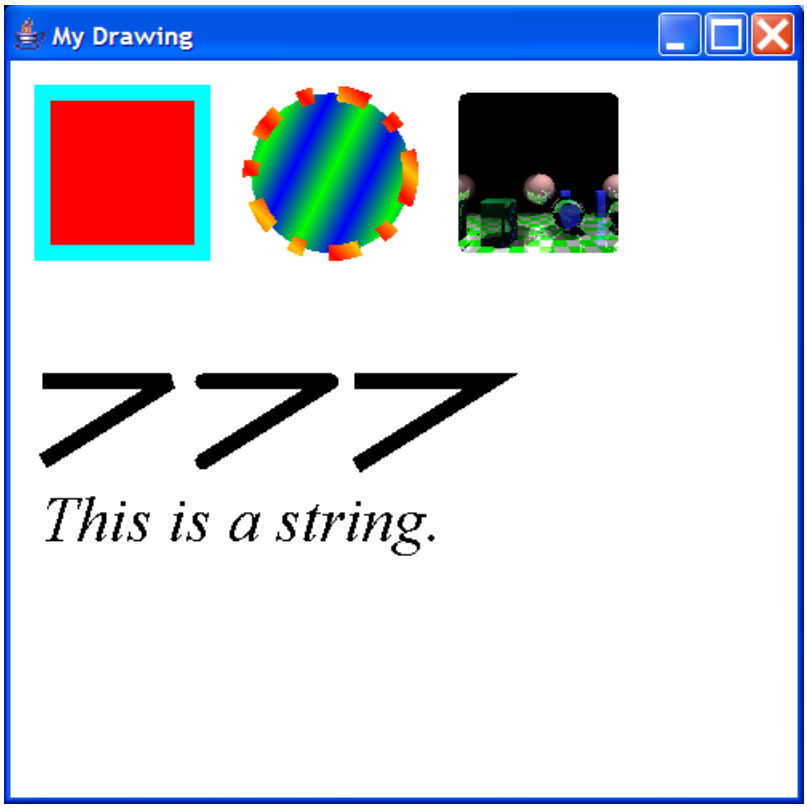


*Figure 10: We have now added in a string. Note that the position (20,300) is roughly at the left bottom of the 'T'.*

not be displayed. By default, the clip is the outer bounds of whatever you are drawing to. There are two forms of setClip. The first takes the coordinates of a rectangular region. The second takes a Shape object. So you could set the clip to be a circle, then draw whatever you want and only the parts that are inside the circle would actually be shown.

The rest of the settings are a bit less obvious, but that doesn't make them less useful. Indeed, they give Java2D some of its more impressive abilities. Normally when you fill a shape you expect the color of that shape to simply overwrite what had been in that location before. It turns out that Java2D supports transparency as represented by an alpha component in the color so this isn't always the case. If you have the paint set to draw with transparency then some of the background color will show through. You can have even more control over this if you want though. The Graphics2D class has a method called setComposite that controls how two colors will be combined. It takes one argument that is of type Composite. There is only one provided implementation of the Composite interface and that is AlphaComposite. It defines a number of static fields for common composite types. The API page for AlphaComposite gives full details of these

and the options you have. For here it is sufficient to say that the color you are drawing doesn't always simply overlay and cover up the color it goes on top of.

So far, every time we have drawn out something we have been drawing to the coordinates of the pixels on the screen or the pixels in the image. Those are called the device coordinates. Prior to Java2D, that was all you could draw to. With Java2D you actually draw to user coordinates and Java2D convert those to device coordinates to do the actual drawing. So how does that conversion take place? The answer is with an affine transformation and the class AffineTransform. An affine transformation is a transformation in which lines that are originally parallel come out being parallel at the end. There are several types of things that you can do to an image that constitute and affine transform. These are translation, rotation, scaling, and shearing. Any combination of these is also an affine transformation and the AffineTransform class has methods in it to help you set these up. The Graphics2D class has a setTransform method that allows you to tell it what transformation you want to do. So consider the situation where you have a square that you want to draw as a diamond. You need to rotate it about its center through $1/8^{th}$ of a revolution.

The real power here is that any of the things we have talked about drawing and things we will mention later can be drawn with a rotation, translation, shear, or scaling to them. That is quite a bit of power. Without this, drawing a string that slants at an angle is a rather challenging task. With the transform that is no problem at all. To demonstrate this, we can add one line of code above the code to draw the string from example 21.5.

```
/**
 * Example 21.6
 * This one line placed before the drawString will rotate the string.
 */
g2.setTransform(AffineTransform.getRotateInstance(0.3,20,300));
```

Instead of having this line come before the drawing of the string, we could have put it at the top of the function so that everything was drawn with that transformation. That would produce the figure below.

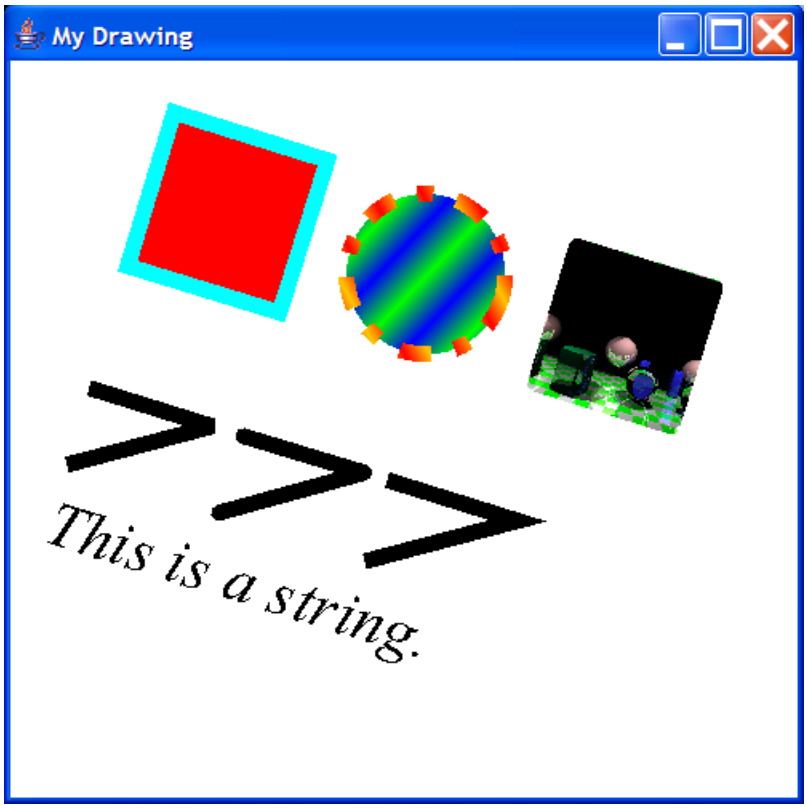


*Figure 12: This figure shows how by moving the transform to the beginning of our rendering we can rotate everything that we have drawn.*

The last setting that we will mention is the ability to give render hints to Java2D. This is done with the setRenderHints method. There are a fair number of hints that you can give to Java2D and we aren't going to go into them in any detail. However, it is worth explaining at least one and letting you know why they are called hints. The render hints are special things that Java could do when drawing to make an image look better or to possibly draw it faster. The reason these are called hints is that Java can ignore you if it determines that what you are telling it to do isn't needed.

A simple example of this is antialiasing. Because images are based on pixels, a straight line in an image or displayed on screen will actually be a bit jagged unless it comes perfectly vertically of horizontally. The process of antialiasing is to turn some pixels part way on based on how much of the line would overlap them. This removes the jagged appearance. It can make some drawings look much better, especially things like fonts. The problem is that it can take time to do antialiasing. If you really care about the quality of your output, you can give Java the hint to do antialiasing. If you need your application to draw really fast though, you might want to turn off the antialiasing.

However, if your machine has a graphics card that can do antialiasing really fast, Java might ignore your hint and do antialiasing anyway. If what you really care about is general speed vs. quality, there is a render hint where you can tell Java to focus on one or the other. The figure below shows the benefits of antialising. Here we have set antialiasing on for the angled lines and the text. The three objects at the top still have aliasing and look a bit jagged as a result.

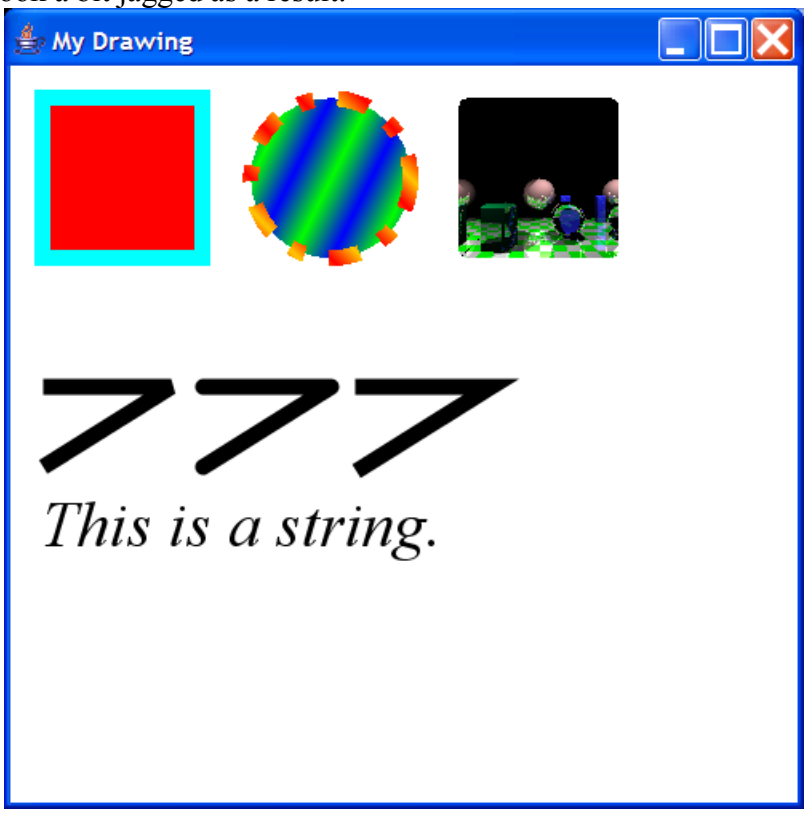


*Figure 13: This figure shows the advantages of turning on antialiasing for making figures look less jagged.*

Graphics2D also provides a number of drawImage methods that allow you to draw an existing image into a Graphics2D object. These can be helpful in many different ways. In fact, most real programs will do what is called double buffering. In this, an image is kept that has what is supposed to be drawn to screen. The paintComponent method does nothing more than draw that image on the the graphics object. What is supposed to be in the image is drawn there before it gets put on the screen. This approach prevents the user from seeing flicker if complex objects are being drawn to the screen. It can also save time if we are drawing something complex and the same thing would be drawn many times on refreshes. This way the complex drawing is done only once and from then on the image is just pushed out to the screen. When images are drawn, the

paint and stroke aren't used, but the composite, transform, and render hints will come into play.

## 22. Full Detail of Exceptions

In section 14 we had a brief introduction to exceptions, just enough to get you through basic Java programming. Now we need to discuss them in full detail. When a function potentially might not be able to do what it needs to, it needs to somehow communicate that to the code that calls it. In C, the way this normally happened was with a return value. For example, if fopen could not open the file for one reason or another, it would return NULL. Some functions can't return invalid values. The atoi function is an example of that. In these situations, some functions will set flags that tell the user information about what went wrong.

The problem with both of these approaches is that coders can easily ignore them in the code and not handle the errors when they occur. This often leads to the error propagating down through the code. Some of the time this becomes a logic error. Other times the program will crash many lines past where the actual error occurs. This is a likely result of what would happen if the programmer doesn't check the return value from fopen. The code won't crash at the call to fopen. It will crash the first time the code tries to read or write from the file, and that could be much later in the execution. The programmer then has to try to track back to the actual source of the error. In C this was even more challenging because the only message you will get when the code crashes is “segmentation fault”.

The fact that these methods of reporting errors make it easy for programmers to ignore the error reports, they allow the errors to propagate, and they provide the programmer with very little information when the code does crash have led more recent languages to use other methods. Java is one of many languages that use exceptions as the primary way of reporting errors. Exceptions correct the primary shortcomings of return codes, but have some challenges of their own.

Exceptions in Java derive from a supertype called Throwable. The Throwable type is the fundamental type for dealing with error conditions in Java, but in practice everything you normally deal with will be a subtype of Exception. There are two basic

types of exceptions in Java: checked and unchecked. An unchecked exception is any class that inherits from RuntimeException. Everything else will be a checked exception. Checked exceptions are impossible to ignore. They get their name because Java checks that you are doing something about them. When you write code that calls methods that can throw a checked exception, you must do one of two things. If that function knows how to recover from the exception, it should include a try/catch statement as described in section 14 that catches that particular exception. There are some situations where the local function won't know what to do to deal with a particular exception. For example if you open a file in a function, but that function doesn't "know" how to alert the user of a failure or do something else to try to fix the problem. In that case, you want to pass the exception up the call stack so that a higher level method will deal with it. In the case of a file, probably the method that actually got the name of the file from the user. In that situation you will need to add a throws clause to the method.

```
/**
 * Example 22.1 – a method with a throws clause
 */
public void openUserSelectedFile(File f) throws FileNotFoundException {
      ...
}
```

There can be several types of exception listed after the throws keyword in a comma separated list. What the throws clause does is to notify methods that call this method that they must either deal with that particular type of exception or else they must also have a throws clause saying they pass it on up.

This requirement of having a try/catch or a throws clause only applies to checked exception. Generally, checked exceptions are supposed to be exceptions that you expect the code truly can recover from and likely needs to put in code to deal with. For example, failure to open a file is an exception that you inevitably need to deal with or else your code will suffer some type of unhappy termination of execution. Not all potential problems in code fit this mold. There are some types of problems that you are very unlikely to be able to recover from. In Java these would be called Errors and they are unchecked. There are also things that can go wrong that generally imply a programmer error and that should lead to controlled crashes to aid in debugging. Some of these problems can also happen so frequently in Java that being forced to include try/catch statements would be extremely burdensome. Examples of these include

NullPointerException and ArrayIndexOutOfBoundsException.

Both of these exceptions inherit from RuntimeException so you can write code that might throw them and you aren't forced to include a try/catch or a throws clause. Since any line involving a dot can throw a NullPointerException, as can any reference to an array, forcing programmers to be deal with them would lead to very messy code. Also, a NullPointerException typically pointers to programmer error and it is something that needs to be corrected in the code, not dealt with at runtime. The same is true for array bounds going out of range.

A big part of the difference is that these exceptions are typically caused by programmer error and not user error. This difference in origin points to a different type of need for handling. Programmer errors should be flagged and crash the code while user errors can't really be “fixed” so instead they should be handled in a reasonable way.

So the checked exceptions simply can't be ignored completely while the unchecked exceptions won't propagate if the programmer does ignore them. This fixes our first two problems with the standard mechanisms for reporting errors. By using the correct exceptions and giving those exceptions the proper information, we can correct the third. That problem is that return types and error flags often don't provide much information for helping programmers correct errors. Code that throws exceptions gets to choose what exception to throw and should pick an exception that describes the nature of the error. More on doing this later. The exceptions themselves are objects so they can hold whatever information is needed to help diagnose the source of the problem. For example, exceptions for indexes being out of bounds should always include information on what the proper indexes are and what index was requested. This helps the programmer know what went wrong without requiring them to put in print statements and try to reproduce the error. Also, when an exception occurs and crashes the code, Java reports the full stack trace. This tells you where the error occurred in the code and where each method was called all the way back up to the top of the call stack.

So what should you do if you are writing code that might not work? An example of this would be in a stack if the user calls pop on an empty stack. Without writing our own code to check for this, the behavior that we get will depend on the implementation. In the array based stack the default behavior would likely be to throw an

ArrayIndexOutOfBoundsException while the stack based implementation would throw a NullPointerException. There are two problems with this. First, those errors are not very informative as they don't do a good job of telling the person using our library what really happened. What is worse, the error behavior changes when the implementation changes. That is something that we really want to avoid because a good programmer would write code to the interface and might catch the exception if it could be caused by user input. An example of this would be if the stack were being used in a reverse polish calculator. It would be easy to see them changing the implementation they use and not noticing that the exception had changed.

The proper behavior here would be for the code in pop to check if the stack is empty and throw a more meaningful exception. Something like a StackUnderflowException would be appropriate, and we could even have our stack interface say that it throws that exception so that the person using the library knows what to expect. At that point, as long as we do a good job of making our library code do what it should, the behavior will be the same for all implementations of the stack and users can switch implementations freely without running into other problems. Also, if the programmer messes up and causes an underflow, the exception will tell them very clearly what they did wrong and make it easier for them to fix the problem.

Example 22.2 shows a possible version of pop from a list based stack that has the proper behavior.

```
/**
 * Example 22.2 – A pop method for a list based stack that throws an exception
 * on underflow.
 */
public E pop() throws StackUnderflowException {
	if(isEmpty()) {
		throw new StackUnderflowException("Attempt to pop empty stack.");
	}
	E ret=head.data;
	head=head.next;
	return ret;
}
```

The throw operator takes a single argument that needs to be of type Throwable. We have created a StackUnderflowException object and we gave the constructor a single argument of a string that gives a description of the error. How do you pick what exception to throw? If something exists in the Java libraries that fits your needs, you should use it. If there isn't an existing class that meets your needs, then you have to create your own. You

define an exception class just like you would any other class, you just need to have it inherit from Throwable. Most of the time you will do this by having it inherit either from Exception or RuntimeException. Which you choose will depend on whether you want the exception to be checked or unchecked.

In the case of StackUnderflowException, we probably want it unchecked as most of the time it won't be caused by user error, but instead by programmer neglect. Also, we probably don't want every call to the pop method to require exception handling. Our StackUnderflowException might look like the code in example 22.3.

```
/**
 * Example 22.3 – A class declaration for an example exception.
 */
public class StackUnderflowException extends RuntimeException {
	public StackUnderflowException(String s) {
		super(s);
	}
}
```

Note that this is a really simple class. This is one of the big advantages of inheriting from Exception or one of its subtypes because we don't really need to add much functionality. Of course, we could certainly add extra data elements into our class if this was an error that had more details associated with it that we needed to report back to the code handling the exception.

There is another advantage of exceptions that doesn't really matter when comparing to C, but does matter for all class based object-oriented languages. This is the fact that constructors can't return anything to tell the user they failed, but they can throw an exception. In fact, the FileNotFoundException is generally thrown by a constructor when the user tries to create an object that uses a particular file.

## 23. Refactoring

One of the more recent additions to the toolbox of programmers is the concept of refactoring. The act of refactoring is to change some aspect of a program without changing the functionality of the program. That last part is extremely significant. If you change what the program does you are not technically refactoring it, you are just making general edits. As you have hopefully learned by this point, software development is a constant battle against complexity. Object-orientation is one of the tools that we use in this battle. It doesn't technically make the software any simpler. What it does is to break

the software into more discrete pieces and give us a framework to think about the different pieces independently. The real advantage is that you can ignore many aspects of a system at any given time in a well designed object-oriented program.

The point of refactoring is also to help us push back complexity. Making changes to large software systems is a delicate task. In fact, the difficulty involved in that task can make many developers very hesitant to make any changes to a piece of software once it gets beyond a certain size. Part of the complexity of making changes comes from the fact that we often try to change what a program does at the same time we change how it does it. With refactoring, you do one, then the other, but never do both together. One significant advantage of this is that if you do a refactoring correctly, the behavior of your program should not change at all and you should be able to test to demonstrate that the functionality is still correct.

It turns out that you do simple refactoring a lot when you are programming. The simplest form of refactoring is nothing more than changing the name of a variable or a method. Granted, once you change the name you must also change all the places in the code that reference it. This can be quite tedious. Many other more complex refactoring are also potentially tedious and can be messed up by humans, but are fairly easy to describe algorithmically. That is why refactoring has been added to a lot of software IDEs, including Eclipse. You might have already noticed that the right click menu often has an option for “Refactor”. What you find under that submenu depends on where the cursor is or what is selected. Rename is nearly always given as an option. Rename changes the name of the class, member, method, or variable, and also goes through the entire project and changes any other occurrences of whatever it is that you are renaming.

As you might guess, there are many types of refactoring other than renaming, but before listing some of them, it is worth discussing when we refactor our code. So the question is, when would you change something in your code when your code works perfectly, and your change won't actually change what the code does? There is more than one answer to this question. The literature on refactoring refers to indicators that you need to refactor as “smells”. As this name implies, you often refactor when there is something that you feel isn't quite right in the code. It isn't blatantly wrong, but it doesn't provide the functionality in quite the way you want it to.

Going back to the example of renaming, simply changing the name of a variable doesn't have any profound effect on your code. However, if the name you picked originally turns out to not be a good reflection of what it is being used for, then the rename operation can be extremely helpful for keeping your code clear and concise in terms of self-documentation. This can save you, or some later programmer working on the code, a lot of time and grief.

Refactoring can also be driven by a desire to add certain functionality. It might be that when you first wrote your program you picked a design for some facet of the program that is not very flexible and now you want to add some functionality to that aspect of the code. Your current design will allow you to make the required addition, but it might be difficult and messy. Even worse, you might now see how this is something that you might be doing again and again in the future. In this case you would want to think of a more flexible design, refactor your code to use that design, then after you have established that the refactoring worked, you could implement the improved functionality in the new design.

So what types of smells are commonly cited as reasons for refactoring code? Here is a list from the book "Refactoring" by Martin Fowler along with a brief description of each. For some of the smells I have listed one or two of the refactorings that could be applied to fix it.

- Duplicate Code – Fairly self-explanatory. The simplest fix is often to extract the duplicated code into a method using the Extract Method refactoring.
- Long Method – You don't want individual methods to be too long because they are hard to work with and often become brittle. One way to fix a method that is too long is to break it up with Extract Method.
- Large Class – Really large classes are just as bad as long methods. This can often be fixed with Extract Class or Extract Subclass.
- Long Parameter List – In an object-oriented program, you don't want to have extremely long parameter lists for methods. When you get one of these, you can fix it by doing something like Introducing a Parameter Object where you pass an object that encapsulates several of the parameters.
- Divergent Change – This is when a single class gets changed often for a number

of different reasons. Ideally each type of change should be made to a different class. You should use Extract Class to pull out the pieces that are altered with each type of change.

- Shotgun Surgery – This is the opposite of divergent change. Here you want to make a single change and find that you have to go and touch a large number of classes to make it. To fix this, you typically use the Move Method and Move Field refactorings to put those aspects that are changed into a single class.
- Feature Envy – This is where a method in one class is constantly pulling data out of another class. In this case you should use Move Method to put the method in the class where it really belongs.
- Data Clumps – This is when you find that there are a number of different data values that seem to be associated with one another frequently in your code. It could be in methods or as parameters to methods. You should use Extract Class to encapsulate these values into their own class.
- Primitive Obsession – This is a common problem for programmers new to object-orientation. They tend to steer away from introducing small classes to hold a few elements of data and instead litter their code with primitives. There are a number of different refactorings where you can replace the primitives with objects and classes to fix this.
- Switch Statements – These were great in C. They are smelly in object-oriented languages. This is mainly because they tend to come in packs and you will have multiple switch statements scattered around that have the same expression and the same cases, but with potentially different code in them. When you add a case to one you are forced to hunt down all the others and add it to them as well. There are several refactorings to prevent this, but the general idea is summed up in Replace Conditional with Polymorphism.
- Parallel Inheritance Hierarchies – This is a special case of Shotgun Surgery because when you edit one hierarchy you have to edit the other as well. To solve this you will use Move Method and Move Field so one hierarchy refers to the other.
- Lazy Class – This is when you have a class that doesn't do much of anything and

needs to be eliminated. In this case you use Collapse Hierarchy or Inline Class to combine it with another class to produce something that carries its weight.

- Speculative Generality – You probably aren't likely to do this right now as it tends to afflict intermediate to advanced programmers with some skill in design and foresight into how code might be changed. The idea is that code is written to be too general because the programmer tried to think of too many things to make it flexible. That's fine if they are being used, but this smell is for when they aren't. In that case the code is likely too complex and it is more difficult to understand than it needs to be. Refactorings like Collapse Hierarchy and Inline Class might be used to trim things down a bit.
- Temporary Field – This is the smell you get when a class has a field in it that is only used some of the time. This is hard to maintain because we expect all the fields to be used all the time. We could use Extract Class to pull the code associated with that field out into a class and potentially Introduce Null Object to remove special case code.
- Message Chains – This is a smell that you likely have in your update methods. It occurs when you have long chains of get methods called on a single line. An example might be loc.getScreen().getBlock(x,y).isPassible(). The problem here is that every time a line like this appears, you have built in a dependency on the specific structure of the chain. The way to correct this is with Hide Delegate or with Extract Method. Using the latter you might write a simple method in your player called getBlock that simply does the loc.getScreen().getBlock(x,y). The advantage here is that if something changed in the way you access a block you would only have to update that one method instead of every single place you used the chain. Granted, this smell is much less pungent if you can be certain the chain isn't going to be changed. That happens to be the case for my example because I'm not going to refactor my infrastructure in the middle of the semester.
- Middle Man – This is a situation where encapsulation goes a bit too far and you have a class that really doesn't do much of anything but delegate to other classes. In this case you can use Remove Middle Man and have direct calls to the class that knows the information you need.

- Inappropriate Intimacy – The basic problem here is that you have two classes that depend too much on the private aspects of the other class. In C++ you could do this with friend classes. In Java it is less likely, but can happen with inheritance because the subclasses always have more access to the superclass than other classes will (or you could be doing something really bad like making members package private or putting in too many get and set methods). You would use a number of different techniques to correct this depending on the nature of the intimacy, but for inheritance you might go with Replace Inheritance with Delegation.
- Alternative Classes with Different Interfaces – This basically means you have two classes that do very similar things, but have differently named methods for doing it. Using Rename Method can help here. If the similarities are strong enough you might even consider going with Extract Superclass.
- Incomplete Library Class – This is the situation where a library class that you can't modify is lacking functionality that you want. You can act like you are adding this functionality with Introduce Foreign Method or Introduce Local Extension.
- Data Class – This is something that programmers often create if they have been programming a while before being introduced to object-orientation. You have a class with data members and each member has a get and a set method, but there is basically nothing else to the class. Worse, the data members might even be public. In that case use Encapsulate Field immediately before you are caught with public data. Either way this is likely to lead to a fair bit of Inappropriate Intimacy so you want to remove all the set methods you can, and move functionality around so this class does more than just act as a place to store data.
- Refused Bequest – This is the smell you get when one class inherits from another and doesn't really take advantage of the functionality or the interface that is provided. Not using all the functionality isn't that bad unless it goes to extreme. In that case you should make a new sibling class and use Push Down Method and Push Down Field to give it the functionality from the parent that you aren't using. Not wanting the full interface is a bigger problem and in that case you should probably apply Replace Inheritance with Delegation.

Once you have identified a particular smell, now you would need to do the appropriate refactoring to fix it. The list of refactorings given by Fowler is significantly longer than the list of smells. So I'm not going to reproduce the full thing here. Hopefully the names of some of the key ones in the description of the smells gives you a good idea. If not, let me know and I'll extend this section to include a longer description of a few refactorings.

Eclipse can help you to refactor your code in a number of different ways. For example, if you highlight a section of a method and right click on it, the refactor menu will give you the option to pull that code out into a new method. This is an application of the "extract method" refactoring and when Eclipse does it you can be certain that the functionality of your code won't be changed. So you can use Eclipse to help you refactor your code in some ways, but it is important to understand what it is doing and why you would want to do some of those things.

## 24. Java Concurrency Library

With Java 5.0 a new library was added to help you do many of the common tasks you will run into when writing multithreaded programs. All of these are tasks can be accomplished using the standard thread capabilities in java.lang, but some of them could be quite difficult to accomplish that way. These are tasks that are done commonly enough that it truly made sense to put them into a standard library. That library is java.util.concurrent.

The heart of the java.util.concurrent package lies in the Executor interface. This is an incredibly simple interface with one method to execute Runnable objects. The key here is that it doesn't tell you anything about how the run method of the Runnable object will be performed. It might be done in parallel in a separate thread, but that doesn't have to be the case. The details of the implementation will determine how the execute method works.

The Executor interface is extremely abstract and you aren't likely to interact with it directly very often. More functionality is provided by the ExecutorService interface, which extends Executor. One of the significant advantages of the ExecutorService is that it provides support for the Callable<T> interface. This is an interface that acts just like

Runnable, only the call method it in has a return value. It returns that generic type given to the Callable. This allows asynchronous calls to do calculations and effectively return them to the main process. The simplest method to start a new task with an ExecutorService is the submit method. There are three versions of this, but we'll focus on the one that takes an argument of type Callable<T>. The problem is that submit can't simply return type T because, like the Executor, we don't know exactly how the ExecutorService is going to wind up invoking the Callable<T> object. It is quite likely it will be an asynchronous invocation, in which case the current thread will continue running before the return value has actually been calculated.

To get around this, the submit method returns an object of type Future<T>. The Future interface represents the result of an asynchronous call that might not have yet completed. As the name implies, it is a way of dealing with the future state of that method invocation. There are two main things you can do with a Future<T> object. The isDone() method allows you to ask if the method has successfully completed yet. The get() method allows you to retrieve the return value of the asynchronous call. If the call hasn't completed yet, the basic get() method will wait and block the current thread until it has finished. This is very similar to what we used join for when we were working directly with threads, only join didn't implicitly support return types.

The ExecutorService interface also has a shutdown method that allows you to tell the ExecutorService to stop accepting new tasks and simply let the existing tasks finish out what they are doing. Other, more advanced methods allow you to schedule multiple tasks with the service in a single call, and other versions of the submit method provide backward compatibility with Runnable. There is also a ScheduledExecutorService that combines some timer like abilities in with the abilities of the basic ExecutorService.

Of course, ExecutorService and ScheduledExecutorService are also interfaces and the exact way that tasks are scheduled is determined by the selected implementation. Most of the time you will not create ExecutorService objects directly. Instead, you will use the Executors utility class. This class has a number of static methods beginning with the word new that create various types of ExecutorServices for you. Each of these provides a slightly different type of implementation for how it manages threads and distributes the various asynchronous tasks.

Of course, when you are running tasks in parallel you need to have abilities to synchronize the tasks and you need data structures that properly support concurrent access. The java.util.concurrent package has support for these things as well. One of the more common types of data structures that can help in synchronizing multiple parallel tasks is a blocking queue. This is a queue in the sense that we have discussed in class, but that only has a certain number of slots in it. It is called a blocking queue because if you try to put something into the queue when there are no slots present, the call will block and the thread trying to add it will have to wait until some other thread takes something out of the queue. Similarly, if you try to remove something from a blocking queue and nothing is on the queue, the call doesn't fail, but instead it waits until another thread adds something onto the queue. This provides a simple mechanism for programmers to deal with certain limited resources in a program. As you might guess, this type of functionality is provided by classes that extend the BlockingQueue interface. Be careful to read the API when you start using this class because there are multiple methods for adding and removing and not all of them block in the way described here, some throw exceptions to signify that an operation couldn't be completed. You will likely want to use the blocking versions most of the time.

To see how a blocking queue could be helpful, imagine a program that processes elements in a manner similar to an assembly line where you have broken the processing task up into several pieces with the goal that different threads can be working on different parts of the processing at the same time. Now extend this to the idea of multiple assembly lines working in parallel. Imagine each assembly line has two parts to it and every piece of data needs go through part A first, then part B. When part A of the line has finished with an object it wants to hand it off to one of the lines in part B. If all the B lines are busy it would be nice if the A line could simply put its product in a bin and work on the next piece of data. Of course, we don't want the two sides getting too far out of synch. We don't want a mountain of partially processed data in the middle if A is running much faster than B and we definitely want B to have a good behavior if it is running faster than A and there isn't anything waiting. Using a blocking queue between the two lines would provide exactly what we want. We could specify how many slots for data there should be in the queue and if an A line finishes a task while those slots are full it

will simply wait until a slot opens up. Similarly, if a B line is ready to start working on another piece and nothing is there, it will wait until something is put on the queue. This form of "assembly line" processing is a standard technique in parallel processing.

The task of keeping different threads well coordinated is one of the most significant issues in multithreaded programming. For this reason, the java.util.concurrent library provides several methods for coordinating threads that go beyond the wait, notify, and join methods you get with basic Java threading. There are four new classes that help with these tasks. They are Semaphore, CountDownLatch, Exchanger, and CyclicBarrier.

A Semaphore is a very standard model used in parallel programming. The idea is that the semaphore allows a certain number of "permits" and threads can either acquire or release these permits. If a thread tries to acquire a permit from a semaphore that doesn't have any more, the thread blocks until one is released by another thread. This is similar in many ways to the blocking queue, but it is simpler, with less overhead, and lacks the ability to attach data to the different permits. There is also no equivalent where too many permits could be given back to the semaphore.

The CountDownLatch and CyclicBarrier are similar classes other than the CyclicBarrier can do the same thing over and over again. These help with the situations where a number of threads are working on a set of tasks and they all have to finish this particular task before they can move on. So the first, second, etc. threads that finish have to wait until the last thread is finished before they can all move on with what they are doing. This type of situation is very common in simulation work where the simulation code has to complete one time step on all the threads before any of them can move on to the next step.

The Exchanger class basically represents a meeting point where two threads communicate. More specifically, they are able to swap objects at the meeting point. The first thread to get to the meeting point will wait until the second has arrived, at which point they swap information and move on with what they are doing.

The new libraries also provide additional container classes beyond the BlockingQueue and can be used in multithreaded code. The basic collections that we have looked on in Java, mainly those that implement List like LinkedList and ArrayList, do not have any type of synchronization. This is because, as we have seen, there is a cost

to synchronization and it should only be used when it is required. However, there are times when multiple threads will need to share access to the same collection. The older collections libraries included methods in the java.util.Collections class that would return synchronized version of various collections that you could use with multiple threads. However, it turns out that these implementations are heavily synchronized, and therefore they are not ideal for code that does a lot more searching and reading through the collections than in does altering it. For those situations, there are newer classes such as CopyOnWriteArrayList and ConcurrentHashMap that will provide better general performance while still being completely safe to share across multiple threads.

There are two other subpackages included in java.util.concurrent. These are java.util.concurrent.atomic and java.util.concurrent.locks. The atomic package provides a host of wrapper classes for different types of data that implement “atomic” access. This use of the word atomic goes back to the Greek origin and has nothing to do with particle physics. The word atomic means indivisible and indeed it was considered for many years that atoms themselves were indivisible. In computing the term atomic is generally applied to operations that can't be interrupted. This is significant in multithreading because there are some things that you have to be able to do without worrying that another thread is going to come in and screw it up in the middle of the operation. Examples of this include things like checking and locking a monitor when a synchornized method is invoked. Imagine if one thread checked the monitor, found it wasn't locked, but before it managed to lock in another thread “snuck” in and did its own check. Now you have two threads that have both seen the monitor unlocked and they both proceed to lock it down and go into the sentitive code where only one thread should be at a time. Obviously, checking monitors needs to be atomic so the process can't be interrupted. The atomic package provides wrappers for ints and other data types that allow you to operate on them with calls that act as if they are atomic so no other thread can come in and mess things up.

The locks package provides what the name implies, ways for code to set locks that prevent other code from getting access. This is like what the synchronized keyword does, but these objects give you more flexibility. For example, synchronized can only lock a single class or object. What if you have sensitive code that spans two different classes or

even just two objects instantiated from the same class, and you need to have it so that no two threads access them at the same time. The synchronized keyword can not do that for you because it only checks the monitor for the current object or class and there is no way to get it to check something that is shared more broadly. The ability to create Lock objects makes this sharing across multiple objects possible. This package also provides a Condition interface that gives you the same type of additional flexibility with regards to the use of wait and notify/notifyAll.

All together, the java.util.concurrent package provides a significant productivity boost to programmers intending to do significant multithreading in Java. This is one of the packages you should likely get a good working familiarity with before you graduate because the importance of being able to understand and write parallel code is only increasing.

## 25. Java I/O

You learned last semester that file I/O can be incredibly valuable for making a program functional for the user. Most all real applications use files in some way, even if it is simply for saving and loading the user's work. Java has a very different system for doing file I/O compared to C. There are good reasons for this, as we will see. One of the things that you might recall from C was that file I/O and standard I/O were very similar. You could use printf and scanf instead of fprintf and fscanf, or you could pass the constants stdout and stdin to the file versions to have them work with standard output and input. In Java you have used System.out a lot and you might have used System.in some as well. It is quite possible that you never really looked at what these objects were, you just know that System.out has print and println methods that allowed you to write things to standard output.

The primary library for doing I/O in Java is in the java.io package. As with most of the packages in the Java libraries, this one includes a fair number of different classes. In the case of java.io, most of those classes are part of a single inheritance hierarchy that is rooted in four base classes. The original base classes in java.io were InputStream and OutputStream. These are abstract classes with a small number of methods. The most important methods are those that read from or write to the stream. In these classes those

methods read or write bytes. There are methods that deal with individual bytes as well as some that deal with arrays of bytes. Since all of the data on computers are stored as bytes these methods are technically sufficient to do anything that you want. They simply aren't very easy to use for a lot of purposes when we desire to read and write higher level constructs and data into files.

These classes are called streams for a reason. The term stream in computing has a very specific meaning. In fact, you have likely heard this term when people talk about media on the internet. Both audio and video that are pulled off remote sources are often referred to as streams. The concept behind the term stream is that data can be pulled off a stream and you can do what you want with it, but you aren't really allowed to jump around and the reading only happens in one direction. Once the data has “passed by”, like the water in a stream, it is gone. Some streams will allow you to use a method called mark to place a marker that you can jump back to, but not all streams have to support this.

The other base classes in java.io are also abstract and they too represent streams, but that term has been dropped from the name in this case. These are the Reader and Writer classes. In many ways these classes are analogous to InputStream and OutputStream, but instead of having methods that work with bytes and arrays of bytes, they have methods that work with characters and arrays of characters.

The fact that these base classes are abstract means that you can't actually instantiate them. There is a good reason for this. The primary method in them, typically read or write, is abstract because it is not specified what type of data source the stream is pulling from. For this reason, there are subtypes of these base types that can pull from different sources. One common type of source or sink for streams is files. If you want to read information from a stream you should use FileInputStream or FileReader. Similarly, if you want to write to a file you would use FileOutputStream or FileWriter. For these streams the read and write methods have been implemented so that they interact with a file. You create objects of these types with different constructors that allow you to specify what file they interact with in various ways. As you might guess, these constructors can throw exceptions in the situation where they don't work. In particular, they can throw a FileNotFoundException. The read and write methods on streams also often have the option of throwing various subtypes of IOException in the case that

something goes wrong with the operation. You'll find that when you are working with files there are quite a few checked exceptions that you will have to deal with. The code segment below shows how we could use a FileOutputStream to write a string out to a file.

```
/**
 * Example 25.1 - A segment of code that writes a string out to a file.
 */
String str="This is a string that we will write to file.";
try {
    FileOutputStream fos=new FileOutputStream("string.txt");
    fos.write(str.getBytes());
    fos.close();
} catch(FileNotFoundException e) {
    e.printStackTrace();
} catch(IOException e) {
    e.printStackTrace();
}
```

Fortunately the String class has a nice method that converts it to an array of bytes so that we can write those out to a file. This isn't the case for everything that you want to write though so we often need classes that have some methods that do more for us than the basic read and write methods do. The way java.io gives this to us is with other subtypes of the stream classes that have extra methods that provide the additional functionality. These classes typically don't specify what type of data source they interact with. Instead, they can be “wrapped” around other stream objects that do actually attach to real data. This wrapping is done at the time of construction of the object because they stream object doesn't have any meaning without a something to read from or write to.

Two classes that we can use to give us additional functionality are DataInputStream and DataOutputStream. We use these classes if we want to read from or write to binary format sources and sinks. Binary format files don't use plain text. Instead they use the same type of format that you have in the memory of the machine. This has some advantages and some drawbacks. The advantages are in speed and size because the files are typically smaller and programs can read and write them more quickly. Unfortunately, they aren't human readable and attempting to edit a binary file with something like vi will generally end very badly. The DataInputStream adds to the basic read methods we get from InputStream by giving us methods to write integers, floating point numbers, and strings.

Some of the streams you can wrap around other streams don't actually add additional methods for functionality. Instead these classes make the standard methods work in different ways. Examples of this are the BufferedInputStream and

BufferedOutputStream. These don't add methods beyond the basic InputStream and OutputStream. Instead, they change the way the that read method works so that bytes aren't requested or written individually. Instead, they are buffered up so that the input and output happen in larger chunks. This type of behavior can greatly improve performance for a lot of different stream types. The next sample shows code that would read an array of doubles using a DataInputStream that has been wrapped around a BufferedInputStream for better performance.

```
/**
 * Example 25.2 - A segment of code that reads an array of doubles from file.
 */
try {
    DataInputStream dis=new DataInputStream(
        new BufferedInputStream(new FileInputStream("array.bin")));
    int size=dis.readInt();
    double[] buffer=new double[size];
    for(int i=0; i<buffer.length; ++i) {
        buffer[i]=dis.readDouble();
    }
    dis.close();
} catch(FileNotFoundException e) {
    e.printStackTrace();
} catch(IOException e) {
    e.printStackTrace();
}
```

Notice the way that the streams are wrapped around one another. This is a style f coding that you will use pretty much all the time when you are working with java.io. This file, because it is a binary file, is not something you can read nicely with standard command line tools and since binary file formats are quite specific, you aren't likely to be able to read it with anything else other than custom written code like this.

There is another set of subclasses of InputStream and OutputStream that are closely related to the DataInputStream and DataOutputStream. These are the ObjectInputStream and ObjectOutputStream. The name implies what they can do. They have all the methods of the Data classes, but they add methods that can actually read and write full objects. At first this might not sound too impressive to you, but if you think about it a bit you'll start to realize how impressive it is. Imagine what you would have to do to write code that could output ANY object and allow you to read it back in later on. That object could be a String, a linked list, a tree, or the screen class you have in your game. Not only is the top level object written, but all the contents that it depends on are written as well. The beauty of the Object streams is that the programmer doesn't have to write code to support any of this.

There are some limitations to what can be written out, but these are for security reasons. You don't want just any code to be able to write your objects to disk as that can provide a lot in insight into the inner workings of the class. People also expect objects written to file to be valid over long periods of time. Having the save format on programs change is a major limitation and you might want to actually change your classes in ways that won't work with writing them to streams.

The limitation is that you can only write objects that are a subtype of the interface java.io.Serializable. This interface has no methods in it so you don't have to actually write anything extra in a class that extends Serializable. Instead, it is just taken to be a flag telling Java that those objects are safe to serialize. As you probably gathered from the wording of the previous sentence, the act of converting an object to a byte stream is called serialization. Java has a default serialization algorithm that uses a tool of Java called reflection that we won't get into, but that you should certainly consider reading up on. This default method isn't ideal for all types of data, for example, a linked list can be written out in the same way an array is without taking up overhead for nodes. The default serialization algorithm will simply write out what is there so you will store a lot of nodes with references between them. There are better ways to do this and you can add methods that override the default serialization algorithm. These are beyond the scope of what we are covering.

I said about that when you serialize an object it writes out the object and all the objects that object references and so on until everything is written. Primitive objects are also written out properly. What happens though if an object contains something that isn't Serializable? That will throw an exception because you aren't allowed to serialize anything that isn't Serializable. If there are parts of an object that you either don't want written out or that you can't write out because they aren't Serializable, you have to make those data elements transient. The keyword transient is similar to final and static in that you can apply it to member data in classes. The meaning of transient is that the transient data types should not be serialized with the object. You have seen this keyword before. In the code I provided you the images are transient. The reason for this is that the various image classes, like BufferedImage, are not Serializable. So if you tried to use the screen editor, which uses Serialization to write objects to file, and you didn't have the images

labeled as transient then the code would throw an exception when it tried to write the images out.

There are other classes in java.io that are useful that aren't part of the streaming hierarchy. One of the most significant of these is the File class. As the name implies, this class can be used to represent a file in the directory system. File objects don't have to refer to files that actually exist. In fact, one of the methods in the File class will tell you if it does exist. Other methods give you information like whether it is a directory, how large it is, permissions, etc. Those are all fairly simple things to do, but they aren't that easy to do in many other languages because they are very platform dependent. Where things get really interesting are the methods of File that you can use on directories to get a listing of all of the files inside of the directory. These types of methods give you a lot of power in your applications, but in most languages they aren't exactly easy to use. Basically, the File class, like the wrapper classes in java.lang, is a really good class to know about because it has a lot of methods that do the various things that you might want to do with a file or give you various information that you might want to know.

## 26. Java Networking

Networking has become as ubiquitous as computing itself today. Most of the time you don't even realize when a program is going across the network for information. You're only truly aware of this if you use a computer that isn't connected to a network and you start to notice there are a lot of things that you can't do. With this global emphasis on networking, it is important for you to know how to write network capable programs. This can be a challenging thing to do in some languages. Fortunately, Java makes it fairly easy for you and all you need to know about is the java.net package.

The most common type of network communication between machines uses the Transfer Control Protocol, TCP. This is a protocol for computers to talk to one another that is dependable. That is to say that if you send a message over TCP and it doesn't arrive on the other side, you will be notified that there was an error. TCP isn't the only communication protocol though and the Java libraries also support the Universal Datagram Protocol, UDP. UDP does not inform you if a message fails to get through. The benefit of UDP is that it can go a lot faster because you don't have to wait around for

verification that your message survived. Granted, it is possible that you will lose a certain fraction of the messages that you send and never know about it if you are using UDP. As a result of this, programs that use UDP have to be programmed slightly differently so that they can recover or even function normally with a certain fraction of the packets being dropped.

In general network communication is done over sockets across ports. We'll look at the details of how we can communication over a TCP connection using the classes in java.net. When two programs want to communicate over TCP, one of the programs needs to designate itself as the server. That program will create a ServerSocket on a certain port. Ports are simply integer values and certain ports are reserved for different standard programs like ftp, telnet, ssh, http, etc. The higher number ports can be used by any program that wants them. Only one program can listen at a given port on a machine at one time.

Once the server program has created a ServerSocket it will call the accept method on that object. This method blocks until another machine makes a connection. Note that in most programs you will probably want to have some type of multithreading going on if you are using networking because otherwise the accept method will make it so that your program can't do anything until another machine connects to you.

To make the connection, the client machine will create a Socket object and tell the constructor what machine and what port to talk to. Once the socket is created on the client and has initiated contact with the server, the accept method on the server returns its own Socket object. So now both the server and the client have socket objects that are connected to one another. These sockets provide the basic abilities for communication between machines. The term socket is actually used for the entities that communicate between machines in all languages, not just Java. In Java, we way we talk across sockets is with streams, just like the ones covered in the previous section. Each socket has a getInputStream() and a getOutputStream() method. These methods return objects of type IinputStream and OutputStream that the applications can write to or read from. All the things you could do with file based streams can be done with socket based streams as well. This includes wrapping them up to give them extra functionality.

This is an incredibly powerful feature and onces again give validation to the

decision to use inheritance and the decorator pattern in the Java I/O libraries. Has all the features you wanted been put in a single file class in Java, all that functionality would have to be duplicated for sockets as well. Instead, we can wrap our socket based streams in buffered streams or data streams, or object streams and get all the power we got with the file based streams without the need to create any extra code. Note that the ability to wrap the socket streams inside of object streams means that we can pass for serializable objects out across the network. This is a tremendously powerful feature and it allows you to do things that would be extremely complex in many languages with great ease in Java.

The UDP sockets work is a rather different way. You must create a DatagramSocket object and attach it to a port. That much is similar. However, the DatagramSocket does not give you input and output streams to play with. Instead it has send a receive methods that both take DatagramPacket objects. The DatagramPacket is a class that can hold a buffer of bytes as well as other information about what is transfered. This means that you must pack up your messages into arrays of bytes on your own and unpack them when they get to the other side. Also, the packets can arrive at their destination out of order. So not only do you have to deal with the possibility that some packets will be lost all together, even the ones that get through could do so in somewhat random order. Basically, there is a lot of overhead for programming using UDP. As a result, only programs that truly need the improved networking speed will go through the effort.

Just like the java.io package had other helper classes such as File, the java.net package has more in it than the various socket related classes. The equivalent to the File class in java.net is the URL class. This is a helpful class that "knows" about HTTP style communication and will handle the lower level networking for you. You create URL objects by giving it a URL. You can then request an input stream for that URL and a connection will be opened to the remove server. This provides a very simple way for you to read files that are stored out across the net. This allows you to make things like web spiders fairly simply and you can even pair it with some of the higher level Swing components to make a simple web browser if you are so inclined. At the very least, the URL class can be used in your assignment if you decide to make it into an applet. Instead of reading images from file, an applet would need to open up a connection back to the

server to pull down the files. The URL class provides a very simple method for doing this.

## 27. Java RMI

The general networking that was discussed in the previous section is flexible enough to do anything you want. Then again, that was what we said about the basic Thread class in Java. Just because you can do everything in a certain way doesn't mean that it is the easiest way to do it. For large network programs, using the basic networking mechanism can be rather complex. To illustrate this, consider a program where the client and the server can exchange ten different types of messages. Each of those messages is going to carry different information and should be directed to a different part of the receiving program. All of this information will need to come across a single socket connection. The code will have to determine what type of message it is reading from the socket and then direct the information to the right part of the program. The first part of that likely means a bunch of if statements or a switch statement. That isn't horrible, but probably isn't ideal either. The second part is more significant because it means that the networking code will have strong dependencies on all the rest of the code and changes to almost anything in the program could require alterations to the networking code.

Of course, one can develop nice, object-oriented solutions to this that minimize the dependencies, but that requires a lot of work that will be very similar for most large networked applications. For this reason, the Remote Method Invocation (RMI) library was created. Remote Method Invocation is exactly what the name implies. It is a set of libraries that allow you to invoke methods on objects that are actually on other machines. The library is in the java.rmi packages and the packages under it. The beauty of RMI is that once you have it set up in a program, but can call a method on a remote object in exactly the same way you would call a method on a local object. The only difference you will notice is that all the remote methods can throw RemoteException so you potentially have more error handling code involved.

There are a few steps that you need to take to get your application to use RMI. First, the methods that you can call remotely need to be specified in interfaces. That is to say that you will create one or more interfaces which list the methods that can be called

across the network. These interfaces will need to extend java.rmi.Remote. Like java.io.Serializable, Remote is an interface that doesn't have any methods in it. As such, it won't force you to add any functionality. All it does is say that when using RMI, objects of this type are to be handled in a special way. You need to make all of the methods in the interface say that they can throw a RemoteException. This is because remote method invocations have to go across the network and might fail if the network connection is broken.

Once you have your interfaces you have to implement them. For each interface you will likely create a class that implements the interface and extends some subtype of RemoteObject. Normally you will extend UnicastRemoteObject. As with any non-abstract class that implements an interface, the implementation must provide code for all the methods in the interface. It can also add whatever other methods it needs to. The methods that are not part of the interface can be used locally, but can not be called remotely. Most of the time in your programs you will refer to the objects by the interface names. If you are referring to them remotely this is required. For this reason, you will probably find that most of the methods you add that aren't in the interface are probably going to be private.

Implementing the class pretty much means that you are done with the coding, but there are still some things you need to do. The problem is that you can't just pull a reference to an object on a different machine out of the air. You have to get these objects from somewhere. One remote object can hand you remote references to many other objects so the real trick is getting hold of the first object. That first object typically has something of a server-like feel to it. You need to make it so that it can be found by code running on other computers to initiate the RMI contact. The way you do this is through the RMI registry. On the “server” machine you will run the program called rmiregistry and leave it running as a background process. This program actually does socket level networking for you and will listen on a specific port for requests. Once you have brought up the registry you need to tell it about the object(s) you want it to be able to hand out. You do this with the java.rmi.Naming class using either the bind or rebind static methods. These methods take a name and an object. The name is can name you want the registry to know the object by and can include a machine specification if you are registering the

object with a registry on a different computer.

Once the registry knows about this first class, often called a factory class because of how it will dispense other remote objects, all that is left is for other machines to be able to find that object and get a remote reference to it. This is done with java.rmi.Naming again. In this case the lookup method is used. It is passed nothing more than a name. Typically this name is a full URL beginning with "rmi://" and giving the machine the registry is running on as well as the name specified for the object. If such an object is registered, lookup will return it. Once this happens, your code can make whatever remote calls you want on the object.

These steps basically tell you want you need to do to get an RMI program up and running, but there is another critical aspect that you need to understand in order to design your RMI program. That is how object are passed around using RMI. As you might guess, passing objects over the network is a bit different than when making normal calls, and because to the significant speed hit taken by talking across the network, you need more flexibility so that you aren't forced to make all your calls across the network.

RMI sits on top of the basic networking we have already discussed and it relies heavily on the object serialization capabilities provided by ObjectOutputStream and ObjectInputStream. When you pass data to a remote method or have a remote method return data, there are three possible things that can happen depending on the nature of the data. If the data is Remote, that is to say it implements the Remote interface at some level, a remote reference to the object will be passed across the network and method invocations will be done across the network. This is the real power of RMI, but it is also really slow in cases where you don't need to be making the calls remotely. If the data isn't Remote, but is either primitive or Serializable, then it is passed by value across the network. This means that the other machine gets a full local copy of the data to itself. Changes to that data have no impact on the original version on the other machine, but they also incur no networking overhead. Most small data objects will be passed in this way.

If you try to pass data that does not fit into one of these categories, you will get an exception. This is because RMI must be able to serialize the data across the network and if an object is of a class that doesn't allow Serialization then you are simply out of luck.

The selection of what to make Remote vs. Serializable and how to deal with data that is neither turns out to be the biggest design challenge when working with RMI. The networking part is simplified to the point that you don't really think about it. It can't disappear completely though and instead you have to think a bit more about the types of data you are passing around and if that makes sense. For large applications this trade off is typically quite acceptable.

The design issues aren't the only trade offs associated with using RMI. As you might guess, there are some speed issues. RMI uses TCP sockets so you simply can't get the speed you could with UDP. What is more, RMI adds overhead in how data is packed up and unpacked. Granted, it prevents you from having to think much about how to do that, but in general you can likely get slightly better network performance if you got through the trouble of writing your own TCP network code with sockets instead of relying on RMI to help you keep things straight.