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 off the knowledge you already have of C. You should read this during the first few weeks of the semester 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 though 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. One 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 ability 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't be seen or used outside 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 of 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 PAD one 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 though some of the details of what some things mean should certainly be unclear.

```
/**
 * Code Example 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 use much 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 are similar and they can only call other static methods and use static data in the class unless they have an instance of that class in them.

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 we see in this example from C 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 and the what should go in the comments for it 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 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 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 an 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 then 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 in 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
 * 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 implementations of classes.

Returning to the code in example 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 suing here. Overloading is when you have two methods that have the same name, but take different numbers of 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 print out the class name and a unique identifier for the object. Notice that in the main of example 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 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 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 languages 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 at the end of this document.

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 files whole contents into 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. These classes provide a number of a number of helpful functions. In the Money class in example 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 string. 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 though they are very powerful and are used extensively in certain types of programming. 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 can be helpful in certain situations. 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 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 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. For example in the main of example 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 the value of if 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 when it comes to the primitive types though you might not notice those differences in normal usage. 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 though, even a long will fail and at that point you can turn to the Java libraries and the java.math package which has classes are 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 0 and 255 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.

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 true 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.

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 Java 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.

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 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 if 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 4
 * 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];
        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 discussion inclusion polymorphism.

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 a lot, 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. 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 would 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 that tells the compiler the exact path to find the code or bytecode at. 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 all 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 is 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 functions that you had in C, 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 the value you give i in one method is not supposed to be remembered for use in another method in most cases.

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 though 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 the 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 you are 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 world 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.

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

with standard Java naming conventions, package names should use only lowercase letters.

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 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 class 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 here I mean that all the data and the code that was associated with the existing class because 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. As 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 give it a B and it will be happy. 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, even then you don't get true subtyping in many ways, like 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

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.

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 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. The 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.

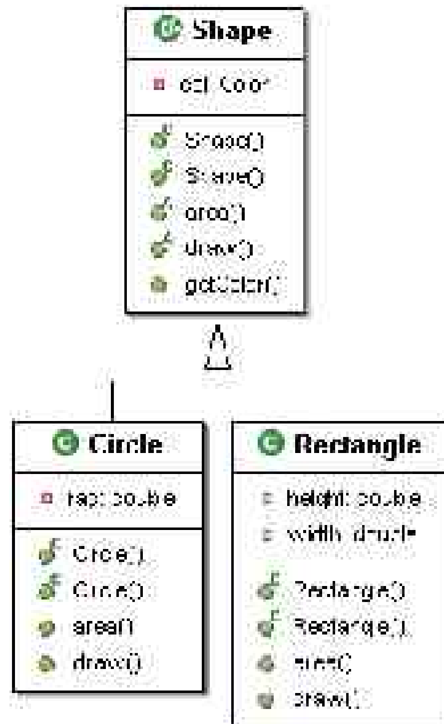

Figure 3 - The class hierarchy for our shape example.

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.

```
/*
 * Example 5
 * 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
```

```java
 */
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 put 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 work 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 5 has two constructors. In the case of the Shape class, it has a **default constructor** as well as a constructor that take a color. The default constructor does not take any arguments, and 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 though that obviously weakens the encapsulation a bit.

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. We already know that's probably a bad idea because we don't want to copy code, and 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 what the subtyping of this inheritance relationship buys us. Example 6 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 6
 * 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 we would do this in C. What would you have to do to do this 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 I'm 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. 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 7 shows a method that includes a try block as well as a catch block.

```
/**
 * Example 7
 * 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 does some type of GUI where the user is supposed to enter in a number if 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.

13. 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 really 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 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 7. 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 7
 * 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;
        }
        public Object data;
        public Node next;
    }
}
```

In this case, the inner class functions more like a struct than a class. You might also notice that I seem to be breaking one of my cardinal rules: data should always be private. The data in the Node class is declared as public. This isn't serious break in encapsulation though because the class Node is itself private so we know it can't be used outside of the LinkedListStack class. Were we to make Node public we would definitely want to change this, but this keeps our code fairly simple while not causing a significant break in implementation hiding or our ability to modify code. Also not that 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 7, 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. Youuse 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 8 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 8
 * 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 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.

This second anonymous inner class in this example is a subclass of the interface Comparator which is used to 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.

14. 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 compilers, 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 sat 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 largely 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.