Java Programming:

1. A simple Java statement ends with a ;

2. A compound Java statement is bracketed by { } and contains 0 or more statements (simple or compound).

3. All Java programming consists of writing non-primitive types.

public class MyFirstClass extends Object {

}

Here is another example, note the need to import JFrame before we can use it.

import javax.swing.JFrame;

public class MyFrame extends JFrame {

}

The part the starts with "public" is called the "class header".

The part between the { } is called the "class body".

The class body is by definition a compound statement, and all elements of the class (fields, method, other classes) go in the body (between the { }).

The parts are:

public: the access modifier

class: this is a class

MyFrame: the name of the class (a name can be almost anything, but professional Java style is to always start with a capital letter)

extends JFrame: indicates the parent class of this class. Every class has exactly 1 parent class.

If you omit the extends part, the class will extend Object by default. Object sits at the very top of the "family tree" of classes in Java and is the only class without a parent.

The access modifier can be any of the four:

public: the code can be used anywhere in the program

package: the code can be used in any class in the same package as the containing class

protected: the code can be used in any class that extends the containing class (or extends the class the extends the containing class, etc.)

private: the code can only be used in the same class

The only code that can go in a class body is:

- field declarations (these can include assignment operators)

- methods

- other non-primitive type definitions (these are called inner types or inner classes)

- initializers: these are rarely used in Java programs and so we might not cover them in the course

4. Each public class must go in its own file. The file name must be the same as the class name, and the file extension must be .java.

The class must be compiled before you use it. The compiler creates a file with the same name but .class extension that contains the Java bytecode for the class.

5. What's the point of a parent class?

- Every class "inherits" all non-private methods of the parent class.

- Every class has access to all non-private fields and inner classes of the parent class.

Thus, we say that "MyFirstFrame is-a JFrame" in that a MyFirstFrame instance can do everything that a JFrame instance can do:

MyFrame frame = new MyFrame();

frame.setSize(300,500);

frame.setVisible(true);

How do we make a MyFirstFrame do more than just a JFrame? We add methods and fields to it.

Methods:

access-modifier return-type name(input parameters) {

}

- here is an example that takes two inputs of type int and returns a value of type int

public int myAddMethod(int input1, int input2) {

- the input parameters are a sequence of 0 or more variable declarations separated by commas. There is one variable declaration for each input your method will take.

- the return type can be any type, or if the method will not return a value, it is "void".

- the part starting with the access modifier is called the "method header"

- the part between the { } is called the "method body".

- all code describing the behavior of the method goes in the body. The body is by definition a compound statement.

public class MyFirstClass extends Object {

public int myAddMethod(int input1, int input2) {

return input1 + input 2;

}

public double square(double x) {

return x \* x;

}

}

A return statement must be included in any method that has a non-void return type. The return statement gives the output of the method.

Now we can use these methods:

> MyFirstClass m = new MyFirstClass();

> m.myAddMethod(300,500)

> 800

> m.square(5.0)

> 25.0

> m.square(5)

> 25.0

When you call the method, Java assigns the first value to the first variable of the input parameters and the second value to the second variable.

All type rules apply. In the last example, the int was automatically converted to the wider double.

> m.myAddMethod(3.0, 5.0)

- ERROR. myAddMethod takes 2 int values, and double is wider than int.

In lab, you saw that some methods could be called with the class name. By default, all methods act on instance of the class. If you want a method that acts on the class itself, you add "static" to the method header.

public static int myStaticAddMethod(int input1, int input2) {

return input1 + input2;

}

Now, let us test:

> MyFirstClass.myStaticAddMethod(3,4)

> 12

> MyFirstClass.myAddMethod(4,5)

- ERROR, myAddMethod is not static

> new MyFirstClass().myAddMethod(4,5)

> 9

> new MyFirstClass().myStaticAddMethod(10,11)

> 21

The last example shows that objects have access to the static methods of their class.

Classes and Types (We only briefly went over this in lecture, but it is an important concept so I am including it here even though we will cover it in a later lecture):

A value of a primitive type is only one type. Either it is an int or a double or a char, etc. Converting a value from one primitive type to another changes the binary representation of the value.

On the other hand, a non-primitive value can be many types at the same time.

Since MyFrame extends JFrame, MyFrame is-a JFrame. Thus an instance of MyFrame is of type MyFrame. It is also type JFrame. (It is also type Object because Object is at the top of the family tree.)

This property is called "polymorhism" for "many types".

For a non-primitive value, we will use the following denotations:

The "true type" of the value is the type it was created as. (The book calls this the "run-time type".) The true type is set by the new operator. For example: new MyFrame() creates an object whose true type is MyFrame.

The true type of a value never changes.

The "current type" of the value is the type it is currently acting as. (The book calls this the "compile-time type".) The current type can be the true type or any type that the true type extends.

We can always determine the current type by looking at the code. For example: JFrame j; Any value stored in variable j will have JFrame as its current type.

Recall that Java will automatically convert a type from a wider to a narrower type.

The same holds for classes. When a class extends another class, the parent class is the wider type and the class that extends it is the narrower type. Classes have to be related to (parent, parent's parent, etc.) for a legal type conversion.

The type conversion changes the current type, but not the true type.

JFrame j = new MyFrame(); <- legal. MyFrame is narrower than JFrame.

MyFrame m = j; <- illegal. j holds the address of a MyFrame, but the type of j is JFrame. JFrame is wider than MyFrame.

MyFrame m = (MyFrame)j; <- legal.

m = (MyFrame)(new JFrame()); <- Never legal. Although MyFrame is narrower than JFrame, the object in question is NOT a MyFrame. All MyFrame's are JFrames, but not all JFrame's are MyFrames.

frame2 = frame; <- Legal. frame1 is type MyFirstFrame and frame2 is type JFrame. The two types are related with JFrame being the wider type.

Finally, note that because the value of a non-primitive type is the location in memory of the instance, type conversions on non-primitive types do not change the values