Chapter 2 Dual Roles of a Class

In the previous chapter, we looked very cursorily at some basic constructs in Java – showing the equivalents of constructs you might have seen in other languages. With this chapter, we will look in-depth at various concepts in Java in particular and object-oriented languages, in general, providing not only their functional description but also their motivation. These concepts make large programs easier but small programs more difficulty to write, much as the formal rules for composing a play makes large Shakesperian plays easier but small tweets and instance messages more difficulty to compose. When we write a play, we create a structure consisting of acts, scenes, paragraphs, sentences and words. An isolated sentence does not make a play but it does make a tweet. Similarly, when we write a Java program, we create a structure consisting of packages, classes, methods, declarations and statements. An isolated statement does not make a Java program but does make a program in some other languages such as Perl, which are designed for writing small programs quickly. Going beyond CS-1 essentially means writing large programs – hence a language such as Java that is designed for such programs is ideal.

Method, Class and Package Encapsulation

To illustrate the emphasis in Java on structuring, consider the following code fragment:

```
long product = 1;
  while (n > 0) {
    product *= n;
    n -= 1;
  }
```

This code snippet does not make a Java program. We must enclose it in a method declaration:

```
public static long loopingFactorial(int n) {
  long product = 1;
  while (n > 0) {
    product *= n;
    n -= 1;
  }
  return product;
```

The method, in turn, must be enclosed in a class:

```
public class Factorials {
 public static long loopingFactorial(int n) {
   …
 }
}
```

The class, in turn, must be put in a package:

```
package lectures.java_basics_overview;
public class Factorials {
….
}
```

Once we have done so, we can access the code in a controlled way from classes in te same or different packages. For instance, we can write the following code to calculate permutations:

```
package lectures.java_basics_overview;
public class Permutations{
  public static long permutations(int n, int r) {
    return Factorials.looping factorial(n) /
                Factorials.loopingFactorial(n-r);
  }
}
```

As we saw in the previous chapter, we must prefix a call to a static method with the name of the class in which the static method is defined , if we are calling the method from a different class. It is possible for two different classes to define methods with the same name. The prefix of a method call indicates the *target* of the method. In the case of static methods, it indicates the class in which the method should be called.

Default Package, Package Declaration Rule, and Package-Level Documentation

A class that does not have a package declaration is put into a default package with no name. When there are multiple classes in the default package, there is no indication of the common problem being solved by the classes. The project name can give some of this information. However, it is not a part of the program that is compiled – it is possible to put the classes in a project with another name without even recompiling the code. Moreover, it cannot indicate the functionality of different sets of classes.

In this course, you will create multi-class programs in almost every assignment. Therefore we will impose the following rule:

> Each class should have a package declaration that describes the problem it solves with related sets of classes.

As we have seen before, this name can be hierarchical, which takes into account that the class may be related to different degrees with related classes. Thus, the package name lectures.scanning of class AnUpperCasePrinter indicates that is first a scanning class and then a lectures example.

Java also allows documentation about a package to be placed in a file with a special name. In the folder created for the package, P, we can create a file called package-info.java. Such a file must have the declaration

```
package P;
```

In addition, it can have comments and annotations describing its function. The following is an example of a package-info file creates in the folder for the package lectures.scanning:

```
/*
 * Code associated with the scanning teaching unit in:
http://www.cs.unc.edu/~dewan/comp401/current/
 * The word and PPT documents created for the unit describe the purpose and
motivation for this unit in
 * great depth
 */
package lectures.scanning;
```

This is the place you should put comments about various parts of your overall architecture. This file has the same short name for all packages - the associated package is selected by putting the file in the appropriate folder.

Class as a Module

Computing factorials is a computationally expensive operation. Hence we do not want to unnecessarily re-compute it. The following code stores a computed value in case we need it again.

```
package lectures.java_basics_overview;
public class StaticLoopingFactorialSpreadsheet {
 static int number;
 static long factorial;
 public static int getNumber() {
  return number;
 }
 public static void setNumber(int newVal) {
  number = newVal ;
  factorial = Factorials.loopingFactorial(number);
 }
 public static long getFactorial() {
  return factorial;
 }
}
```

This class declares two static global variables, number and factorial, which store an integer and its factorial, respectively. These variables are not declared as public. This means they cannot be accessed from classes in other packages. Hiding this information apparently reduces programming flexibility, as arbitrary classes cannot access the variables to create new functionality. Yet, every code rule-book will tell you that variables should not be made public. Why?

The main reason is that programmers of a class are responsible for maintaining the integrity of the class and improving its functionality. Other classes are expected to know only how to invoke the services offered by the class – they are expected to be oblivious to implementation details. How some logical data are represented internally in a class is an implementation detail – and it is best not to expose the representation to arbitrary classes lest they use it to violate the integrity of the class and prevent its evolution.

To illustrate, in our example, the value of the variable, factorial is expected to always be the factorial of the value of the variable, number. By not making these two variables public, we prevent an arbitrary un-trusted class from violating this constraint. The following code shows an alternative version of the body of this class in which the factorial is not stored in a variable - it is computed whenever the getFactorial() method is called.

```
 static int number;
 public static int getNumber() {
   return number;
 }
 public static void setNumber(int newVal) {
   number = newVal ;
 }
 public static long getFactorial() {
   return Factorials.loopingFactorial(number);
 }
```

This version executes faster or slower – that is more or less *time efficient* - than the previous version based on how many times we call getFactorial() after a setNumber() . Moreover, this version uses less space – that is it is more *space efficient* - as it defines one rather than two variables. So we might want to shuttle between these implementations, based on the needs of the users of the class, without worrying about changing other classes. By not making the representation public, we limit the set of other classes that have to be touched when we change the class. Thus, by reducing the set of ways in which other code can interact with a class, we increase the number of ways in which we can evolve the class – a reduction in flexibility of one kind results in an increase in flexibility of another kind.

We see through this discussion a reason for dividing our program code into units – to create walls around these units so that only certain aspects of the units are visible to other units. A program unit that controls access to the variables and methods declared in it is called a module. Thus, a class is a module.

Least Privilege and Non-Public Variables

The principle that governs what should be exported is the least privilege principle which applies to situations beyond programming:

> No entity – human or computational - should be given more rights than it needs to do its job.

This is also called the “need to know” principle, as it Implies that an entity should be given rights to only those objects (physical or computational) about which it needs to know.

In the context of programming, this means a piece of code should not be given more rights than it needs to do its function.

We almost always want the ability to change the variables defining the representation of a class and force integrity constraints associated with these variables. Thus, we rarely want to make these variables public. Thus, follow what other rulebooks have told you:

> Do not create public variables.

There will be times when you feel no harm can come from making them public - it is more than likely that your rationale is based on a shortsighted view of how the program will change.

This rule, of course, does not preclude named constants from being made public. Moreover, creating public classes and main and other methods might put you in the habit of adding the word public to all of your declarations. The principle of least privilege says that think before you add this keyword to any declaration and never add it to a variable declaration.

Bean Conventions Properties

We saw two classes above that seem to have the same functionality. In particular, they provide a repository of a pair of related values – a number and its factorial. These values form the external state exported by the classes. This state is to be distinguished from the internal state composed of the variables of the class. As we see in the second version of the class body, part of the external state, the factorial, is not even stored in an internal variable. The external state is defined by the headers of the public methods of the class. By using a set of standard conventions, called the Bean conventions, it is possible to automatically determine the external state of a class, without relying on subjective interpretation of the class headers. In the rest of this material, we will assume these conventions. We will refer to each unit of the state exported by a class as a *property.* Like a global variable, a property may be static or not. For now, we focus on static properties.

A class defines a static property P of type T if it declares a getter method for reading the value of the property, that is, a method with the following header:

public static T getP()

If it also declares a setter method to change the property, that is, a method with the header

public static void setP (T newP)

then the property is *editable*; otherwise it is *read-only*.

As we see from these definitions, the getter and setter methods of a property must begin with the word “get” and “set”, respectively. Of course, names do not affect the semantics of these methods. For instance, had we named getFactorial, as obtainFactorial, we would not change what the method does. However, in this case, we would be violating the Bean conventions for naming getter and setter methods. The words “get” and “set” are like keywords in that they have special meanings. While keywords have special meanings to Java, “get” and “set” have special meanings to those relying on Bean conventions. Under these conventions, the names of both kinds of methods matter, but not names of the parameters of the setter methods.

On the other hand, the number and types of parameters and results of the methods matter. The getter method must be a function that takes no parameter, while the setter method must be a procedure that takes exactly one parameter whose type is the same as the return type of the corresponding getter method.

These conventions, like any other programming conventions, are useful to (a) humans trying to understand code so that they can maintain or reuse it, and (b) to tools that manipulate code. A class that follows these conventions is called a *Bean*.

These are only one set of conventions you should follow. You have seen others before, such as the case conventions for identifiers, and you will see others later.

Bean conventions were developed in the context of Java – hence the name (coffee) "beans." Some subsequent languages – in particular C# - put even more emphasis on properties by providing language constructs to replace the conventions.

Based on above definition of properties, each of the two implementation above defines one editable (static) property, Number, and a read-only property, Factorial. As mentioned before, the properties defined by a class are related to but not the same as the instance variables of the class. In the second version, the property, Number, is stored in the instance variable, number, but the property, Factorial, is not associated with any instance variable. As also mentioned before, the difference between properties and instance variables is that the former are units of the external state of an object, while the latter are units of the internal state of the object.

A property whose value does not depend on any other property is called an *independent* property, and one that depends on one or more other properties is called a *dependent* property. These two kinds of properties correspond to cells associated with data and formulae, respectively, in an Excel spreadsheet. Hence the suffix "spreadsheet" in the names of the classes presented here.

In this example, both independent properties are editable, while the dependent property is read-only. In general, however, it is possible for an independent or dependent property to be either editable or read-only. Consider a Factorial class customized for a particular number. In such a class, the number would never change. Thus, there would no need to make this property editable. Moreover, in another version of the Factorial class, it could be useful to make the Factorial property editable, and when it is set to a new value, and automatically calculate the value of the number whose Factorial is closest to the one set, and then re-computes the factorial.

Multiple Number-Factorial Pairs

Each of the two versions of the class above allows us to define a number factorial association. What if we wish to keep multiple number factorial associations at the same time? For each of these associations, we could create a copy of the class. However, this approach has the problem of copying code we mentioned in the first chapter, and perhaps more important, does not allow new associations to be created dynamically, while the program is executing. A more practical approach is to create a new class that stores the numbers, and possibly also the factorials, in arrays. However, the size of the array should be large enough to accommodate all desired associations, which may not be known. Moreover, this approach requires more tedious programming involving arrays. Perhaps most important, it is not possible to deal with individual associations as independent units that can be passed as parameter values in methods. For instance, we may wish to define a method that prints the association. We do not have a way to type of the value. Some languages define records or structs to solve this problem, but

these types allow their components to be changed in arbitrary ways - they do not, for instance, prevent the number factorial association from being changed by the print method.

Class as a Type

The solution to this problem is to treat a class not just as a module but also a type describing an infinite set of values or instances of the type. This means that each instance should be able to store an independent set of values in variables declared in the class. The static global variables we have seen earlier are created once for each class – hence the name static. What we need are a different kind of variables, called instance variables, that are created for each instance of the class. By simply omitting the word **static** in the declaration of a class global variable, we declare an instance variable. An instance variable cannot be accessed by a static method, as it is not clear which copy of this variable should be referenced by the method. It can be accessed by an instance method, which belongs to an instance rather than a class and can access the instance variables defined by the class. An instance method is declared by simply omitting the word **static** in the header of the method. An instance method can access a static variable as there is no ambiguity about which copy of the variable should accessed – there is only one copy of such a variable.

Figure 1 shows how we can use these concepts to convert the (first version of) StaticLoopingFactorialSpreadsheet to another class, ALoopingFactorialSpreadsheet, that defines multiple number factorial associations.

```
public class ALoopingFactorialSpreadsheet {
 int number;
 long factorial;
 public int getNumber() {
  return number;
 }
 public void setNumber(int newVal) {
  number = newVal ;
  factorial = Factorials.loopingFactorial(number);
 }
 public long getFactorial() {
  return factorial;
 }
}
```

This class is identical to StaticLoopingFactorialSpreadsheet except that all static keywords have been omitted.

Using this class is more complicated. We cannot use the class the target of a method defined in the class. Thus, the following is illegal:

ALoopingFactorialSpreadsheet.getFactorial()

This is because, getFactorial is not a static method.

Before we can invoke an instance method in this class, we must create an instance on which the method must be called. To create a new instance of a class, C, we invoke the **new** operation, which takes the class name (strictly a "constructor" as we see later) as an argument. The operation returns a new value or *object* whose type is the class, C. As a result, this value can be stored in a variable, c, of type C:

C c = new C();

Each time a new object is created, a new copy of the instance variables of the class of the object are created, which are manipulated by an instance method, m, of the object:

c.m(<actual parameters>)

The following code illustrates how we create new instances of a class and invoke instance methods in them:

```
ALoopingFactorialSpreadsheet factorial1 =    new ALoopingFactorialSpreadsheet ();
ALoopingFactorialSpreadsheet factorial2 =    new ALoopingFactorialSpreadsheet ();
factorial1.setNumber(2);
factorial2.setNumber(3);
System.out.println(factorial1.getFactorial());
System.out.println(factorial2.getFactorial());
```

Here, the two invocations of the new operations create two different copies of the Number and Factorial variable. The two invocations of the setNumber() method manipulate these two copies respectively. As a result, the two println() invocations output the factorial of 2 and 3, respectively. We can create an arbitrary number (limited by size of computer memory) of instances of ALoopingFactorialSpreadsheet.

Thus, with instance variables and methods, a class becomes more than a module, it serves also as a type. Unlike a type such as int or double, which is predefined by Java, it is an example of a type defined by us. A type added to the programming language is called a *programmer-defined* or *user-defined type*.

Instance Properties

Just as the headers of the static methods of a class define properties of the class, the headers of the instance methods of a class define various instance properties of the instances of the class. Thus, a class defines a instance property P of type T if it declares an instance getter method for reading the value of the property, that is, a method with the following header:

public T getP()

If it also declares an instance setter method to change the property, that is, a method with the header

public void setP (T newP)

then the property is editable; otherwise it is read-only. The definitions of dependent and independent and stored and computed instance properties are analogous to those of corresponding static properties.

Static vs. Instance Named Constants

It is also possible to declare a named constant without the static keyword:

final int FEET_IN_YARD = 3;

However, as the value of this "variable" does not change, it does not make sense for Java to create a separate copy of the named constant for each instance. However, Java currently does not allow an instance named constant to be accessed by static methods. Therefore, for more flexibility, it is best to declare it as a static constant.

static final int FEET_IN_YARD = 3;

BMI Spreadsheet

The following is another example of a programmer-defined type, which describes a BMI property that depends on a Weight and Height property.

```
public class ABMISpreadsheet {
        double height;
        public double getHeight() {
                return height;
        }
        public void setHeight(double newHeight) {
                height = newHeight;
        }
        double weight;
        public double getWeight() {
                return weight;
        }
        public void setWeight(double newWeight) {
                weight = newWeight;
        }
        public double getBMI() {
                return weight/(height*height);
        }
}
```

As in the case of ALoopingFactorialSpreadseet, we can now create and manipulate one or more instances of ABMISpreadseet :

```
ABMISpreadsheet bmiSpreadsheet = new ABMISpreadsheet();
bmiSpreadsheet.setHeight(1.77);
bmiSpreadsheet.setWeight(75);
double computedBMI = bmiSpreadsheet.getBMI();
System.out.println(computedBMI );
```

This code instantiates ABMISpreadsheet, next invokes setter methods in the instance to give values for height and weight, then calls a getter method to retrieve the computed dependent BMI value, stores this value in a local variable of type ABMISpreadsheet, and then prints the associated BMI value.

Hiding Fields, Scopes, and This

Suppose we named the formal parameter of setWeight() as weight:

```
public void setWeight(double weight) {
        weight = weight;
}
```

The formal parameter hides the instance variable as it is declared in a scope – a block of code in which declarations can be made – nested in the scope in which the instance variable is declared. Thus, the assignment statement simply reassigns the formal parameter to its value.

There are many ways to fix this problem. The keyword, **this,** references the instance on which the method is declared and we can use it as a target of an instance variable specification as in:

```
public void setWeight(double weight) {
        this.weight = weight;
}
```

However, it is possible to make the mistake of not using **this**. One way to catch this mistake is to declare the formal parameter as final, as we have no intention of changing it in the method.

```
public void setWeight(final double weight) {
        this.weight = weight;
}
```

It is (almost) never a good idea to change values of formal parameters as we are then using them for a purpose they were not intended and violating the rule of assigning an identifier a single logical role. Ideally, Java should have implicitly made all of them final so we do not have to go through the effort of inserting this keyword and making our headers longer. Many style rules as for these parameters to be declared final, but in this material, we will not follow this excellent rule to make the headers more compact. However, we will use the following rule:

Do not assign a value to a formal parameter.

We can make follow a convention of naming instance and local variables differently. One convention that many follow is to prefix the name of an instance variable with the letter m:

```
double mWeight;
```

This letter stands for “member”. This rule was derived in the context of C++ in which instance variables and instance methods of a class were called *members*. We also will use this definition of member in this material.

If such a names looks ugly to you, the one I oftenfollow is to use the prefix “a” for a non-instance variable:

```
public void setWeight(double aWeight) {
```

```
        weight = aWeight;
}
```
Think of the instance variable as holding the definitive values of data while local variables holding temporary values (which go away when the scope declaring them stops executing) – hence this convention. For setters, a simpler rule is to always call the formal parameter, newValue or newVal:

```
public void setWeight(double newValue) {
        weight = newValue;
}
```
Such a convention makes it easy to create a setter for a new property by copying one for another property – the name of the parameter of does not have to changed. As mentioned earlier, the fact that we copy setters is a flaw with the design of Java that C# as fixed. Eclipse will generate setters for you but the generated setters , unfortunately, hide names of instance variables.

instanceof

The Java **instanceof** operation makes the idea of class instances explicit. Given an object o and class C:

o instanceof C

is true if o is an instance of C.

Thus, given the declarations above:

(new ABMISpreadsheet()) instanceof ABMISpreadsheet → **true**
(new ABMISpreadsheet()) instanceof ALoopingFactorialSpreadseet → **false**
bmiSpreadsheet instanceof ABMISpreadsheet → **true**

Constructors

It is possible for instance variables to have illegal values. Consider the object we get when we instantiate ABMISpreadsheet. Non positive values of height and weight do not make sense. Worse, the value of BMI that is calculated from these values is not a number. Of course, as we saw before, after instantiating the class, we could call the setter methods to give legal values to its instance variables:

```
ABMISpreadsheet bmiSpreadsheet = new ABMISpreadSheet();

bmiSpreadSheet.setHeight(1.77);

bmiSpreadSheet.setWeight(75);
```

Java, in fact, allows us to combine these steps in one statement, resulting in a much more succinct version of code that does not even require us to declare the variable bmiSpreadsheet:

```
ABMISpreadsheet bmiSpreadsheet = new ABMISpreadsheet(1.77, 75);
```

It allows us to define in a class a special method, called a constructor, which is automatically called after an instance of the class is created, and returns the instance after possibly initializing its variables. The above scheme for instantiating ABMISpreadsheet requires us to add the following *constructor* to class ABMISpreadsheet:

```
public ABMISpreadsheet(

double theInitialHeight, double theInitialWeight) {

setHeight(theInitialHeight);

setWeight(theInitialWeight);

}
```

The constructor takes as argument the desired initial values of the height and weight and assigns them to the corresponding variables by calling the appropriate setter methods. We could have directly initialized the variables in the constructor using assignment statements:

```
public ABMISpreadsheet(

double theInitialHeight, double theInitialWeight) {

        height = theInitialHeight;

        weight = theInitialWeight;

}
```

However, it is a good idea to invoke a setter method to change a variable. The reason is that if we find that the variable has an illegal value, we need to trace only the setter method in the debugger to determine what is wrong. Otherwise we have to locate and trace all assignment statements that change the variable.

The declaration of a constructor is a little different from other methods. We have no freedom in choosing its name - the name of a constructor must be the same as the name of the class in which it is defined. This name also serves as the type of the object returned by the method, which is the newly created instance after it has been initialized by the constructor. As a result, the header of the constructor does not specify an explicit type name. In this example, ABMISpreaedsheet is both the name of the constructor and name of the type of the new instance returned by the constructor. If we accidentally put a type name in the header, Java will not recognize the method as a constructor:

```
public ABMISpreadsheet ABMISpreadsheet(

double theInitialHeight, double theInitialWeight) {

height = theInitialHeight;

weight = theInitialWeight;

}
```

A constructor exists in every class. If we do not explicitly declare one, Java automatically adds an "empty" default constructor, that is, a constructor with no parameters that does no initialization. For instance, since we did not add to ABMISpreadsheet any constructor, Java automatically added the following constructor to it:

public ABMISpreadsheet () { }

We do not actually see it in the class declaration, since is added to the compiled code rather than the source code. The body of the method does nothing as there are no variables to initialize.

Thus, when we created an instance of ABMISpreadsheet without the constructor:

new ABMISpreadsheet()

we actually invoke its default parameter-less constructor.

Since ABMISpreadsheet now defines an explicit constructor, it no longer includes the default parameter-less constructor. As a result, the above way of instantiating ABMISpreadsheet is no longer valid. If we wish to allow both schemes for instantiating the class, we can explicitly add the parameter-less constructor to the class:

public ABMISpreadsheet() {}

The class now has two constructors with the same name but different sets of parameters:

public ABMISpreadsheet ABMISpreadsheet**(**

double theInitialHeight**, double** theInitialWeight**) {**

height = theInitialHeight;

weight = theInitialWeight;

}

public ABMISpreadsheet() {}

Thus, like ordinary methods, constructors can be overloaded.

Necessary Initializing Constructors and Immutable Objects

The example of ABMISpreadsheet shows that instance variables can be initialized using setter methods or programmer-defined constructors. The advantage of using programmer-defined constructors is that they make the code more compact and ensure that objects are never unininitialized. However, such constructors are not necessary to use the full functionality of the class. For certain kind of classes, however, they are necessary. These are classes whose instances are immutable, that is, cannot be changed after they are instantiated. Such classes, thus, do not define public methods to change the

state of their instances. Thus, the state of the instances must be initialized using programmer-defined constructors when they are created.

An important example of an immutable object is a String, which cannot be changed after it is constructed. Therefore, the class defines a constructor to specify the initial value:

```
String s = new String("hello");
```

The statement above is what the following statement translates to:

```
String s = "hello";
```

A String instance cannot be changed after it has been instantiated. In general, immutable objects make it easy to write programs that use threads and hashtables, which you will study later. This is the reason why String instances are immutable. Java provides another class, StringBuffer, to store mutable strings. Thus, if a string you need is never changed, make it an instance of String so that programs that use it are easier to write; otherwise make it an instance of StringBuffer.

Reassigning Variable vs. Changing Object

To better understand the notion of an immutable String and object variables, compare the following the following code:

```
String s = new String("hello");
s =  s + " world";
```

with the following:

```
StringBuffer s = new StringBuffer("hello");
s.append(" world");
```

In both cases, the we have appended " world" to "hello". However, in the first case, we did not change the (String) instance storing "hello" - we created a new instance that consists of the original string concatenated with the string "world" and assigned this object to the variable holding "hello". Thus, we reassigned a new object to the variable. In the second, we actually change the (StringBuffer) instance holding "hello" by invoking on it a method that appends " world" to its current value.

In general, if you need to a changed version of some object, it is more efficient to invoke on a method on the object to change its state, if such a method exists, rather than creating a brand new object that contains the new version.

Objects vs. Primitives

In the code above, bmiSpreadsheet is an object variable since it stores an object – an instance of the class ABMISpreadsheet. In contrast, computedBMI is a primitive variable, since it stores a double. Values of type int, char, and double are not objects. Non objects are called primitives as they are used to build objects. For example, the class ABMISpreadsheet was implemented using doubles. The type of a primitive and object is called a primitive and object type, respectively.

When a primitive value is assigned to a primitive variable, the value is copied to the memory slot created for the variable. When an object value is copied to an object variable, the address of the object , called a pointer, is copied to the variable. There are several reasons for this difference. One of them is that size of the slot created for the object variable does not have to be the same as the size of the object assigned to it. Thus, objects of different sizes can be assigned to an object variable. When we study interfaces and inheritance, we will see advantages of this flexibility.

Primitive types are also present in other object-oriented programming languages such as C++, but some more pure object-oriented languages such as Smalltalk only have object types. Primitive types can be more efficiently implemented than object types, which was probably the reason for including them in Java. However, this benefit comes at the expense of ease of programming and elegance.

Wrapper Classes

The difference between primitives and objects is directly understood and somewhat overcome by wrapper classes. A wrapper class is associated with each primitive type. For example, the class Integer is asspcoated with the primitive type as int. It defines the "object" version of the primitive. Like any other class, such a class can be instantiated. For example, the following code

```
Integer  ii = new Integer(4)
```

creates an Integer object for the int 4. It is possible to extract an int value from a wrapper, as in:

```
int i = ii.intValue();
```

In general, a wrapper class:

- provides a constructor to create a wrapper object from the corresponding primitive value,
- stores the primitive value in an instance variable,
- provides a getter method to read the value.

We will see later that wrapper objects are crucial for storing primitive value in variables that expect objects.

The wrapper classes for the other primitive types, double, char, boolean, float, long, and short are Double, Character, Boolean, Float, Long and Short.

public Long(long value);

public long longValue();

public Short(short value);

public short shortValue();

Storing Primitives Values and Variables

Consider how the assignment:

int i = 5;

is processed by the computer. From a programmer's point of view, of course, the variable *i* gets assigned the value *5*. However, let us look at under the hood and see what Java exactly does when processing this statement.

Java creates an integer value, 5, and stores it in a memory *block*. A memory block is simply a set of contiguous memory cells that store some program value. It also creates a memory block for the variable *i.* When the assignment statement is executed, the contents of the value block are copied into the variable block.

The two blocks have the same size because the value and the variable have the same type. As a result, the value "fits" exactly in the variable.

The statement:

double d = 5.5

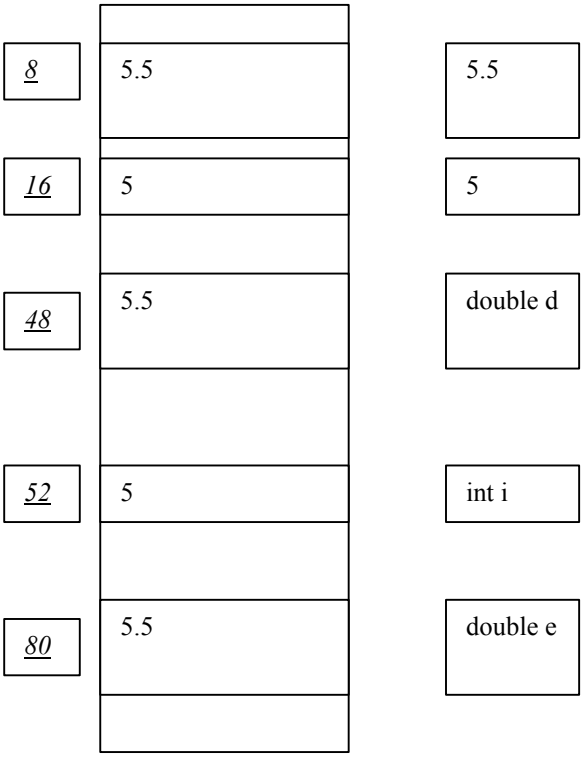
is processed similarly except that Java manipulated blocks of 2 words instead of 1 word, because the size of a double is 2 words.

Assignment of one variable to another is handled similarly:

double e = d;

The contents of the RHS (Right Hand Side) variable of the assignment are copied into the block allocated for the LHS (Left Hand Side) variable.

The following figure illustrates this discussion.



Each memory block is identified by its underlined memory address, which is listed on its left in the figure. Thus the address of variable I is 52 and the literal 5 is 16. While we think in terms of high-level specifiers such as i and 5, the processor, in fact, works in terms of these addresses. The compiler converts these names to addresses, so that the human and processor can speak different languages. It is for this reason that a compiler is also called a translator.

Storing Object Values and Variables

Object values and variables, however, are stored differently. Consider:

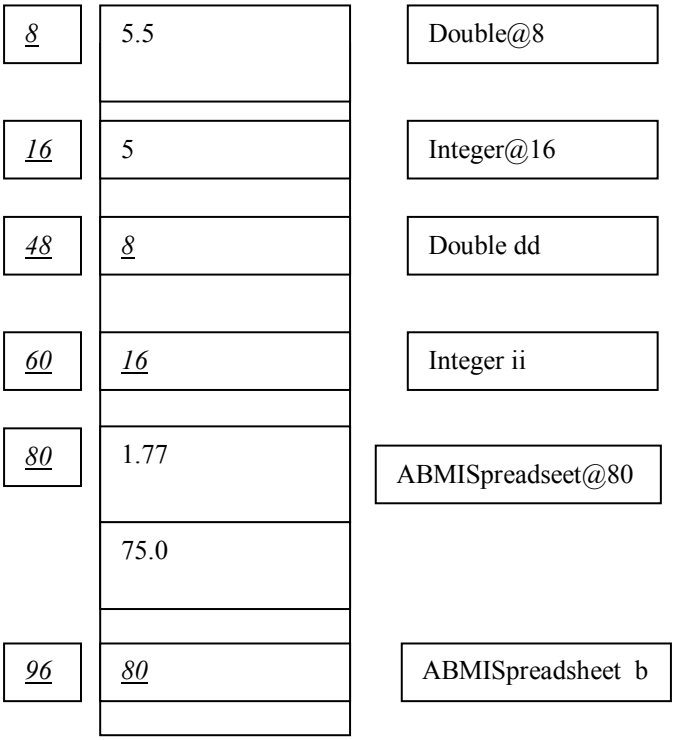
```
Integer ii = new Integer (5);

Double dd = new Double(5.5);
```

As before, both values and the variables are allocated memory. However, each assignment copies into the variable's block, not the contents of the value block, but instead its address. All Java addresses are 1 word long, so all variables are allocated a 1-word block, regardless of their types. Thus, both the `Double` variable, `D`, and the Integer variable, `I`, are the same size, which was not the case with the `double` variable, d, and integer variable, `i`, we saw above.

All objects, however, are not of the same size. When a new object is created, a composite memory block consisting of a series of consecutive blocks, one for each instance variable of the object, is created. Thus, assuming an `Integer` has a single `int` instance variable, a block consisting of a single integer variable

is created. Similarly, for a `Double` instance, a block consisting of a single `double` instance variable is created. The sizes of the two objects are different because of the difference in the sizes of their instance variable. However, in both cases, the object block consists of a single variable. In the case of ABMISpreadsheet, the memory block consists of two double variables, weight and height. The figure below illustrates object storage.



The figure below also shows how the following assignment of an instance of this class is processed:

ABMISpreadsheet b= new ABMISpreadsheet(75, 1.77) ;

The address 80 of the memory block created for ABMISpreadsheet is assigned to the variable.

String as a Predefined Object Type

Class ABMISpreadsheet is both a programmer-defined and object type. Not all object types are programmer-defined type. A String type is an example of a predefined object types. Thus, if we do not initialize a String variables, and invoke a method such as length() on it, we will get a NullPointerException , as in the code below:

```
String s;
System.out.println("String length:" + s.length());
```

== vs equals()

To better understand String and objects, consider the following code:

```
static final String PRESIDENT_NAME = “John F. Kennedy”;
…. // omitted method header and other code
String s = Console.readString();
If (s == PRESIDENT_NAME )
        System.out.println(“s ==  PRESIDENT_NAME”);
String s2 = s;
If (s == s2)
        System.out.println(“s ==  s2”);
```

The first check will never succeed, regardless of what the user inputs. The reason is that the instance returned by Console.readString() is different from the one assigned to PRESIDENT_NAME. The == operation checks if its left and right operand contain a pointer to the same instance. The second check will succeed as the pointer stored in s is assigned to s2. Thus, both variables point to the same instance.

What if we want to check if two String instances represent the same sequence of characters. In this example, what if we want to check if the user input the character sequence “John F. Kennedy.” We can invoke the equals() method on a String, which determines if the target String is equal to the parameter of the method. Thus, we can rewrite the first check as:

```
If (s .equals(PRESIDENT_NAME) ) {
        System.out.println(“s .equals(PRESIDENT_NAME)”);
}
```

What if we never initialized s? We would get a NullPointerException as no method, including equals(), can be invoked on the null value. The equals() method, on the other hand, can take a null value as an argument. Thus, in this example, it is better to write:

```
If (PRESIDENT_NAME).equals(s) ) {
        System.out.println(“s .equals(PRESIDENT_NAME)”);
}
```

as no exception is reported this time and the program works correctly. In general:

> *If a String literal or named constant is to be compared to a String variable using equals, make the former the target of the method and the latter the argument to the method.*

Default Values for Objects and Primitives

Let us consider a variation of the BMI spreadsheet code above in which neither the object nor primitive variable has been initialized, that is, assigned a value explicitly.

```
public class BMISpreadsheetUser {
    public static void main(String[] args) {
        ABMISpreadsheet bmiSpreadsheet;
        bmiSpreadsheet.setHeight(1.77);
```

```
        bmiSpreadsheet.setWeight(75);
        double computedBMI;
        System.out.println(computedBMI );
    }
}
```

Every variable has a value, which is the content of the memory slot assigned to it. This means that if the program does not explicitly assign a value to a variable, it has a default value, which is essentially a 0 in each binary bit of the memory slot. How this value is interpreted depends on the type of the variable. For a numerical primitive variable, it translates to the 0 value. For a char variable, it translates to a character whose integer code is 0, which is called the null character and is represented as '\0'. For Boolean variables it translates to the choice whose representation is 0, which is the **false** value. For object variables, it translates to the **null** pointer, that is, an address that points to nothing. Recall that when we assign an object to a variable, we assign the address of the object to the variable. If we assign nothing to the variable, then it contains the **null** value. We can assign this value explicitly to a variable :

bmiSpreadsheet = **null**;

and also check if the value of an object variable is null:

```
if (bmiSpreadsheet == null) {

    System.out.println ("variable is  null");

}
```

The default values of primitive variables are legal primitive values. As a result, it is legal to involve operations on them, as in:

computedBMI*2

On the other hand, the default value of an object variable is an illegal object value in that it is illegal to invoke operations on it, as in:

bmiSpreadsheet.getBMI()

If we invoke a method on an uninitialized object variable, we get a NullPointerException.

Static and Instance Members Mixed

We have seen above two different implementations of a factorial spreadsheet – one that uses only static members - variables and methods – and one that uses instance members. It is possible and useful to combine both instance and class members, as shown in the following class definition.

```
public class ALoopingFactorialSpreadsheet {
  int number;
  long factorial;
  static int numInstances;;
```

```
 public ALoopingFactorialSpreadsheet){
    numInstances++;
 }
 public int getNumber() {
  return number;
 }
 public void setNumber(int newVal) {
  number = newVal ;
  factorial = Factorials.loopingFactorial(number);
 }
 public long getFactorial() {
  return factorial;
 }
 public ALoopingFactorialSpreadsheet getNumInstances() {
   return numInstances;
 }
 public  Factorial averageAloopingFactorialSpreadseet aFactorial1,
          AoopingFactorialSpreadseet aFactorial2) {
          AloopingFactorialSpreadseet result = new AoopingFactorialSpreadsheet:
          Result.setNumberaFactoria1.getNumber() + aFactorial2.getNumber())/2;)
          Return result;
          }
Public static void main (String args[]) {
    AoopingFactorialSpreadseet aFactorual 1  = new  ….
   AoopingFactorialSpreadseet aFactorual 2 = new  ….
aFacorial1.setNumber(2);
aFactorial2.setNumber(5);
AoopingFactorialSpreadseet result = ALoopingFactorialSpreadseet.average(factorial1, factorial2);
System.out.println(result.getFactorial());

}
```

This class declares both a static variable, numberOfFactorials, and three static methods, getNumInstances (), average()., and main(). The static variable is keeps the number of instances created by the class and the corresponding getter returns this value. A new instance of the class is created each time we use the new operation on the class. We cannot trap this operation, but we can do is define our own constructor that is called after the instance is created by new. This constructor can then increment the static variable.

The average() operation takes two factorlal spreadsheets and returns a new factortial spreadsheet whose number property is the average of the number properties of the two arguments. Its factorial property is of course the factorial of the average. It first creates an new spreadsheet and then sets the number to be the average of the two arguments.

The main() method is a simple testing method to see if the class is working properly. A more elaborate testing method would also test negative and large values.

(Exercises: make class immutable. Should it define two or one variable for time efficiency? define a constructor that does not require setting of the average. Later, define interface for spreadsheet and use interface typing rule. Change static and instance variables. Negative numbers. Maxint)

Real-World Analogy

To better understand Java objects, we can draw an analogy between them and physical objects such as rooms, doors, walls, windows, etc. Since program objects are created by human beings, they are more like manufactured physical objects, such as cars and bicycles, rather than natural objects such as trees and rocks. We interact with a (manufactured) physical object by performing different kinds of operations on it. For instance, we accelerate, brake, and steer a car. The set of operations we can perform on the car is determined by the factory that manufactured or *defined* it. Defining a new kind of car, thus, involves constructing a new factory for it.

Table 1. Computer vs. real-world

Computer-world	Real-world
Class	Factory
Computer Object	Manufactured Physical Object
Method	Operation
Invoking/Executing a Method	Performing an Operation
Instance of a Class	Manufactured by a Factory
Defining/Declaring a Class	Constructing a Factory
Instantiating a Class	Manufacturing an Object

Similarly, we interact with a program object by performing different kinds of operations or methods on it. The methods that can be invoked on an object are determined by the *class* of the object, which corresponds to the factory that defines the blueprint of a manufactured physical object. Defining a new kind of computer object, then, involves creating or *defining* or *declaring* a new class for it. Just as a car can be manufactured on demand by its factory, a computer object can be created on demand by instantiating its class. The reason for choosing the term “class” for a computer factory is that it classifies the objects manufactured by it. Two objects of the same class are guaranteed to have the same behavior, much as two cars produced by the same car factory are expected to work in the same way.

This analogy between Java and physical objects can be used to better understand the difference between static and instance variables/methods. An instance variable is like the mileage record kept in a particular car. Each car has a separate copy of the record. An instance method is like a “getMileage” operation “invoked” on a car to display instance-specific data. A class variable is state kept with the

factory itself such as the number of cars produced by the factory, how long it has been in business, the car operating instructions. A class method is an instance-independent operation, which can retrieve the state invoked on the factory or, for instance, produce a new car whose color is a blend of the colors of existing cars . A class such as the StaticLoopingFactorial that has no instance variables is like a service station – it produces no products but has state such as gas remaining and operations such as buy some of the remaining gas. As this analogy shows, it makes sense for a class to have both instance and class variables/methods, which is what we saw in the last example.