COMP 110/401

Prasun Dewan[1]

4. State

The computations performed by the code we wrote in the previous chapter were essentially the kind we would do with a calculator. The difference was that our code was tailored to a particular kind of calculation. While users of a calculator are responsible for entering the entire formula that is to be computed, the users of our classes were responsible only for entering the data – the formulae were burnt into the code. We essentially learnt how Java can be used to define the kinds of functions some calculators allow us to program.

A computer is, of course, more than a programmable calculator. Here we will see another, related use of it – to implement custom spreadsheets, which, like the custom calculators we saw before, have formulae burnt into them, requiring the users to enter only the data.

If you are familiar with how to program a calculator or a spreadsheet, Java might seem much too complicated for programming these examples. Languages tailored to a particular kind of computation are bound to be simpler than a general-purpose language such as Java. The purpose of these introductory chapters is to use these simple domains to explain aspects of programming that will be crucial for implementing more complex domains, which we will see later.

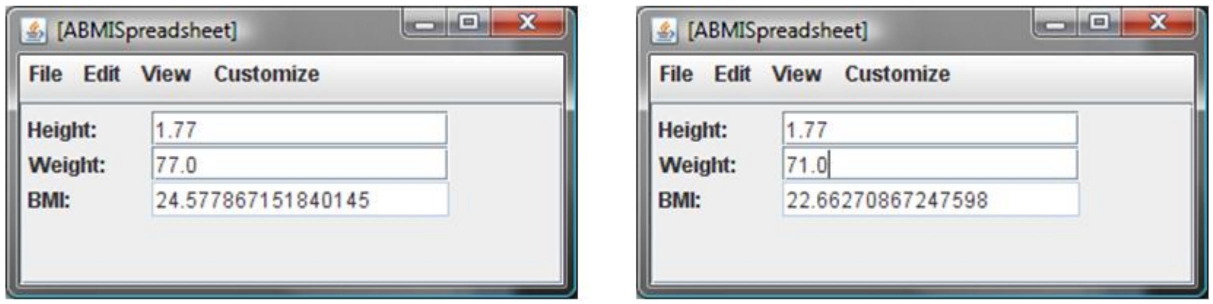
In this chapter you will learn about object *state*. In general, the state of a program component is information remembered by the component, which influences the behavior of the component. For example, the state of a method consists of data stored in its formal parameters, which influences the computation performed by the method.

The state of an object, on the other hand, is information remembered by the object, which persists between method invocations. We will see how to create object state, how to implement spreadsheet-like constraints among different parts of this state, and how to make part of the state of an object accessible to classes other than the class of the object. You will also learn how to trace a program using print statements.

Instance Variables

Consider performing "what-if" BMI computations. In particular, given our height, we may wish to see what the BMI would be for different values of our weight. As we saw in chapter 3, when we use a

[1] © Copyright Prasun Dewan, 2009.



Height: 1.77
Weight: 77.0
BMI: 24.577867151840145

Height: 1.77
Weight: 71.0
BMI: 22.66270867247598

Figure 1. "What-if" computations with ABMISpreadsheet

general purpose BMI calculator, we are forced to enter the same value of the height each time, as shown in the figure above.

As we also saw in chapter 3, one way to address this problem is to create a new specialized class that defines the following method to compute the BMI.

```
public double calculateMyBMI(double weight) {
        final double MY_HEIGHT = 1.77;
        return (new ABMICalculator()).calculateBMI(weight, MY_HEIGHT);
 }
```

This method takes a single parameter, weight, representing the changing weight. The value of the unchanging height is stored in the named constant (final variable), MY_HEIGHT. This method instantiates ABMICalculator, and then calls the calculateBMI method to perform the bmi computation.

The specialized BMI calculator solves the problem of re-entering height. However this approach hardwires our height in the code. This means we must create a separate class for each user who wishes to perform such what-if calculations!

In general, the approach of creating a general-purpose calculator for evaluating a formula is not suitable for what-if computations, since each time the formula is to be evaluated, we must enter all the data items on which it depends. What we need is a spreadsheet-based approach, shown in Figure 1. Here, the weight, height, and BMI are shown as form items in an ObjectEditor window. To try a different value of weight, all we have to do is edit the old value, and press Enter. The object uses the new value of weight and the old value of height to re-compute and refresh the value of the BMI form item. We can try other values of weight in the same fashion. Thus, instead of re-entering all the data of a formula, now we must re-enter only the item we wish to change.

As it turns out, coding a spreadsheet-like program requires a fundamentally different approach to programming – the kind of programming for which object-based languages are particularly well suited. In the examples we have seen so far, the computation performed by a method depended entirely on the parameters provided by the user when it was called. This is exactly what we wish to avoid now. We wish the computation to depend on the state of the instance on which the method is invoked, that is, data remembered by the object. The data must then be stored in variables of the instance that are not specific to a method call – such variables are called *instance variables.*

To illustrate, assume that the new class we wish to create is `ABMISpreadsheet`, and the new method for calculating the BMI is called `getBMI`. Figure 2 shows the difference between `calculateBMI` and `getBMI`. The body of the former accesses its parameters; the body of the latter accesses instance variables of a `ABMISpreadsheet` instance. As Figure 2 shows, unlike method parameters, instance variables are not contained in a method and belong to the whole instance rather than a specific method. For this reason, instance variables are called *global variables* and formal parameters are called *local variables*.

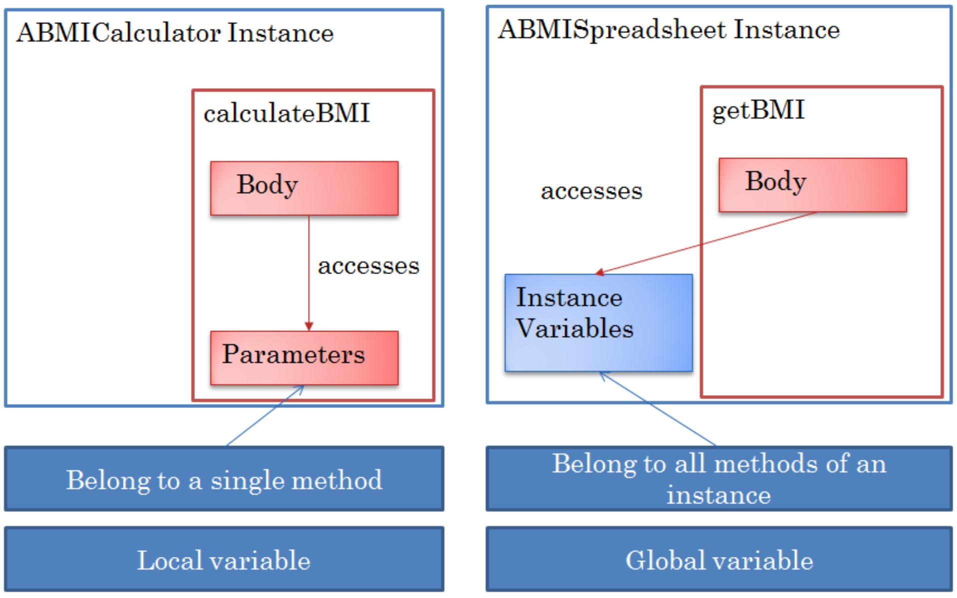


Figure 2. Formal parameters vs. instance variables

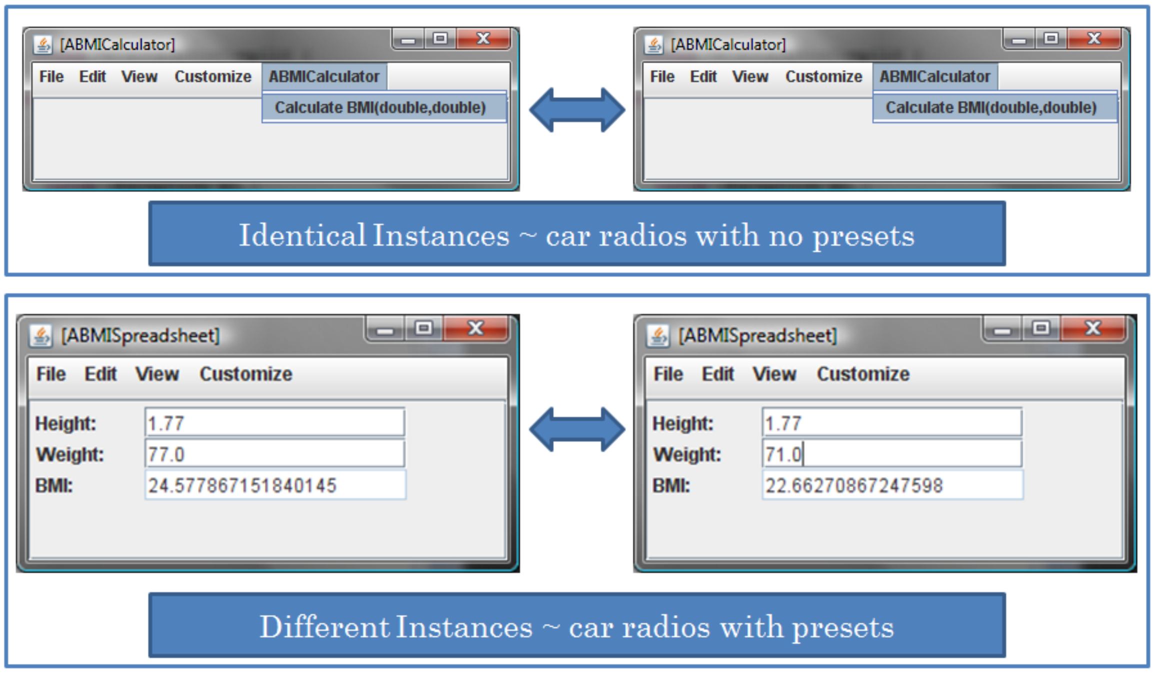

[ABMICalculator]
File Edit View Customize ABMICalculator
Calculate BMI(double,double)
[ABMICalculator]
File Edit View Customize ABMICalculator
Calculate BMI(double,double)
Identical Instances ~ car radios with no presets
[ABMISpreadsheet]
File Edit View Customize
Height: 1.77
Weight: 77.0
BMI: 24.577867151840145
[ABMISpreadsheet]
File Edit View Customize
Height: 1.77
Weight: 71.0
BMI: 22.66270867247598
Different Instances ~ car radios with presets

Figure 3. State-full vs. state-less objects

Instance variables of a class distinguish one instance of a class from another. The reason is that different instances of a class may have different values stored in their instance variables, causing the methods invoked on these instances to return different values. For example, one instance of `ABMISpreadsheet` may store the value 77.0 in the instance variable, `weight`, while another may store 71.0. The method, `getBMI`, would return different values when invoked on the two instances.

Instances of a class that does not declare any instance variables cannot be distinguished from each other. For example, all instance of `ABMICalculator` behave the same, because the class does not define any instance variables.

Data stored in the instance variables of an object is called the *state* of the object. Thus, the state of an object is information remembered by the object between invocations of methods on it.

Objects without state, that is, without instance variables, are analogous to car radios without presets. It does not matter which of these radios we use, because each of them is identical to the others. Objects with state are like car radios with presets. It matters which of these radios we use, because different stations may have been associated with the presets.

The following outline of class `ABMISpreadsheet` illustrates how instance variables are declared and accessed.

```
public class ABMISpreadsheet {
        double height;
        …
        double weight;
        …
        public double getBMI() {
                return weight/(height*height);
        }
        …
}
```

The declaration of an instance variable is much like the declaration of a formal parameter or in that it consists of the name of the variable preceded by the name of its type. However, while a formal parameter is declared inside (the header of) a method, an instance variable is declared outside the methods of the class. Variables declared inside methods are called local variables and other variables are called global variables. Local variables are accessible only in the methods in which they are declared while global variables are accessible to all methods in the class.[2] A local variable may be a formal parameter or a variable such as declared inside the body of a method. MY_HEIGHT, declared in the method calculateMYBMI() above, is example of a local variable declared inside the body of a method.

The method `getBMI` illustrates the global nature of the two instance variables, accessing them to compute its return value. In fact, unlike the other functions we have seen so far, it is a parameter-less function, accessing only instance variables.

[2] Strictly speaking, an instance variable is accessible only to the "instance methods" of the class. The methods we have seen so far are all instance methods; later we will see other kinds of methods called "class methods".

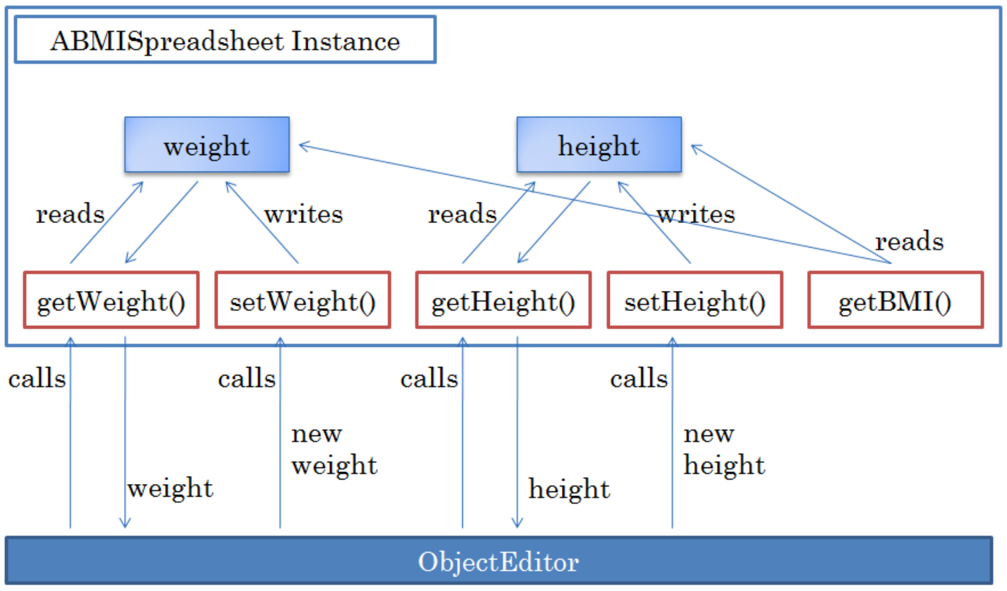


Figure 4. "What-if" computations with ABMISpreadsheet

Getter and Setter Methods

If the values of the two variables are set, `getBMI` will return the correct result. But how do we set the values of these variables?

Figure 1 shows the user-interface for setting the values. ObjectEditor creates slots for these two variables, displays the current values of these variables, and allows the user to edit these values to enter new values for the variables. For this user-interface to work, we need a way for ObjectEditor to get and set the values of the two instance variables.

As it turns out, ObjectEditor is not allowed to directly access the two instance variables, which are not accessible from classes other than `ABMISpreadsheet`.[3] It can, though, do so in a roundabout way, if the class defines appropriate methods. The variables are accessible to methods declared in `ABMISpreadsheet`. Moreover, ObjectEditor can call public methods of the class. Thus, if the class defines methods to read and write the variables, then ObjectEditor can call them to get and set these variables. Methods that retrieve values of instance variables are called *getter methods*, while those that change their values are called *setter methods*.

Figure 4 illustrates the nature of these methods. `ABMISpreadsheet` defines the getter method, `getWeight`, which returns the current value of the instance variable, `weight`. It also defines the setter method, `setWeight`, which takes as a parameter the new value to be used for `weight`, and stores this

[3] We will look in-depth at the reason for this restriction later – is analogous to one that prohibits drivers of cars, who are not trusted to be as competent as factory engineers, from tinkering with the engines if they wish to keep their warranties.

Some languages make the difference between a function and procedure more clear. For example, in Pascal, a function declaration begins with the keyword, `function`, and a procedure declaration begins with the keyword, `procedure`.

Java has adopted the C philosophy of not requiring the programmer to explicitly distinguish between the two kinds of methods, letting the absence of a valid return type indicate a procedure. This approach has the advantage that it saves on the amount we type. It also shows how functions and procedures evolved.

Computer scientists first created scripts to mimic mathematical functions. They then realized that it is useful to code scripts that do not return a value, and hence invented procedures. Thus, a procedure was considered a special kind of function that does not return a value.

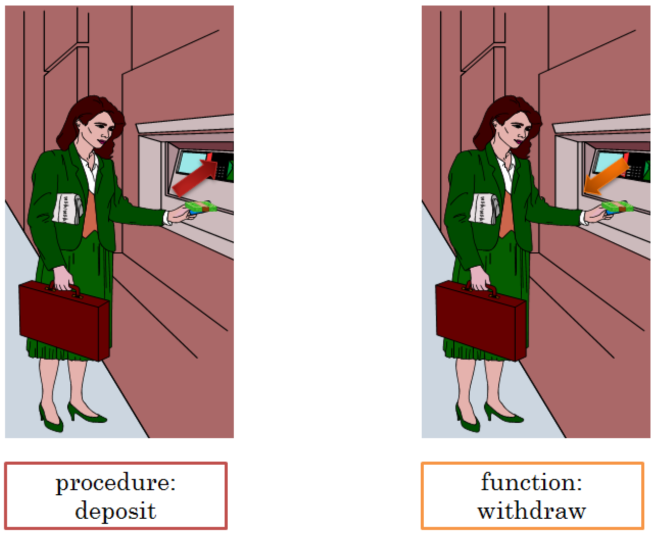


Figure 5. Real-world analogy for functions and procedures

value in the variable. Similarly, it defines `getHeight` and `setHeight` as the getter and setter method, respectively, for `height`. ObjectEditor can, thus, call these four methods to read and write the values of the two instance variables.

Function vs. Procedure

Note that while getter methods are like the methods we saw earlier in that they return values, the setter methods do not return results; instead, they simply assign their parameters to instance variables. Methods that do not return values are called *procedures*, while those that do are called functions, as mentioned in the previous chapter. Thus, the difference between the two kinds of methods is that a function performs a sequence of steps that compute a return value, whereas a procedure simply performs a sequence of steps, without returning back any information.

Let's use a real-world analogy to distinguish between these two kinds of methods. Consider the operations to withdraw and deposit money (Figure 5). The former is a function, returning money after it finishes execution, while the latter is a procedure, simply following a sequence of steps without giving anything back in return.

Figure 6 shows the nature of a procedure declaration and how it differs from a function declaration. The getter method `getWeight` is a function, and hence, is declared like the functions we have seen before, specifying a return value and the type of the return value. The setter method,

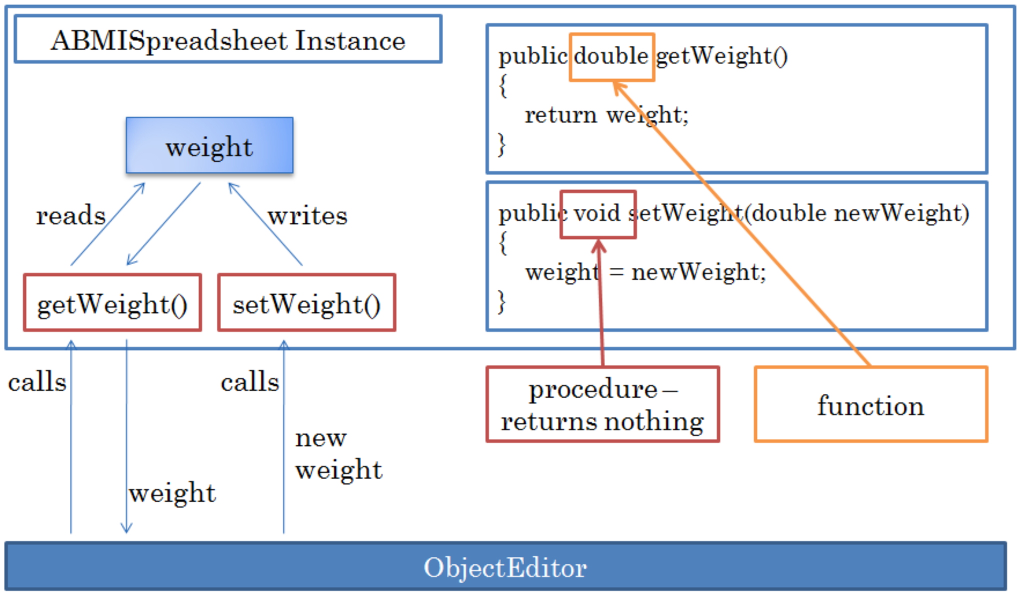


Figure 6. Coding functions vs. procedures

`setWeight` on the other hand, is declared differently, because it is a procedure. The method header does not contain the name of a return type; instead, it contains the keyword **void** to indicate that the method does not return any kind of value. Because no value is returned, there is no return statement in the method.

What is the use of calling a method that does not return a value? As this example shows, such a method can essentially "deposit" a value in a global variable, to be retrieved in the future by a function, much as an ATM transaction can deposit money to be withdrawn by a future transaction.

Let us now look at the complete code of `ABMISpreadsheet`.

```
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);
        }
        }
```

The getter methods do not need any explanation since they are functions, which we have seen before. The setter methods, of course, are different, because they are procedures, returning no values.

Assignment Statement

Instead of return statements, the setter methods contain a different kind of statement, called an *assignment statement*. An assignment statement is a statement that has the assignment operator "=" (equals sign). The part of the statement to the left of the operator is called the *left hand side* (LHS) of the statement, while the part to the right is called the *right hand side* (RHS) of it. The LHS must be a variable and the RHS an expression, that is, a piece of code that yields a value. Thus, the statement has the form:

```
<variable> = <expression>
```

Here, the two words delimited by `<>`, `<variable> and <expression>`, are placeholders for actual program entities. The effect of an assignment statement is to store the value produced by the RHS expression in the LHS variable. We saw the concept of assignment earlier when we discussed the assignment of actual parameters to formal parameters of a method, which takes place implicitly when the method is called. We also saw this idea when we assigned initial values to named constants and variables in their declarations. An assignment statement allows the programmer to explicitly assign values multiple times to arbitrary variables.

Thus, the effect of the assignment statement:

```
height = newHeight;
```

is to assign to the instance variable, `height`, the value stored in the formal parameter, `newHeight`. Essentially, this statement stores the value of one variable in another variable, much as the "Save As…" command stores the contents of one file in another file, or the copy and paste commands together store the text at one location in another.

Figure 7 shows what exactly happens when the `setHeight` procedure is called with the actual parameter, 77.0, illustrating how an (implicit and explicit) assignment works, and the difference between a formal parameter and an instance variable.

Step 1: Before the method is called, there are two variables in memory, the two instance variables, `height` and `weight`. Let us assume their values are 0.0.

Step 2: When the method is called, Java creates a new variable in memory for the formal parameter, `newHeight`, of the method.

Step 3: Next, it assigns the actual parameter, 77.0, to the formal parameter.

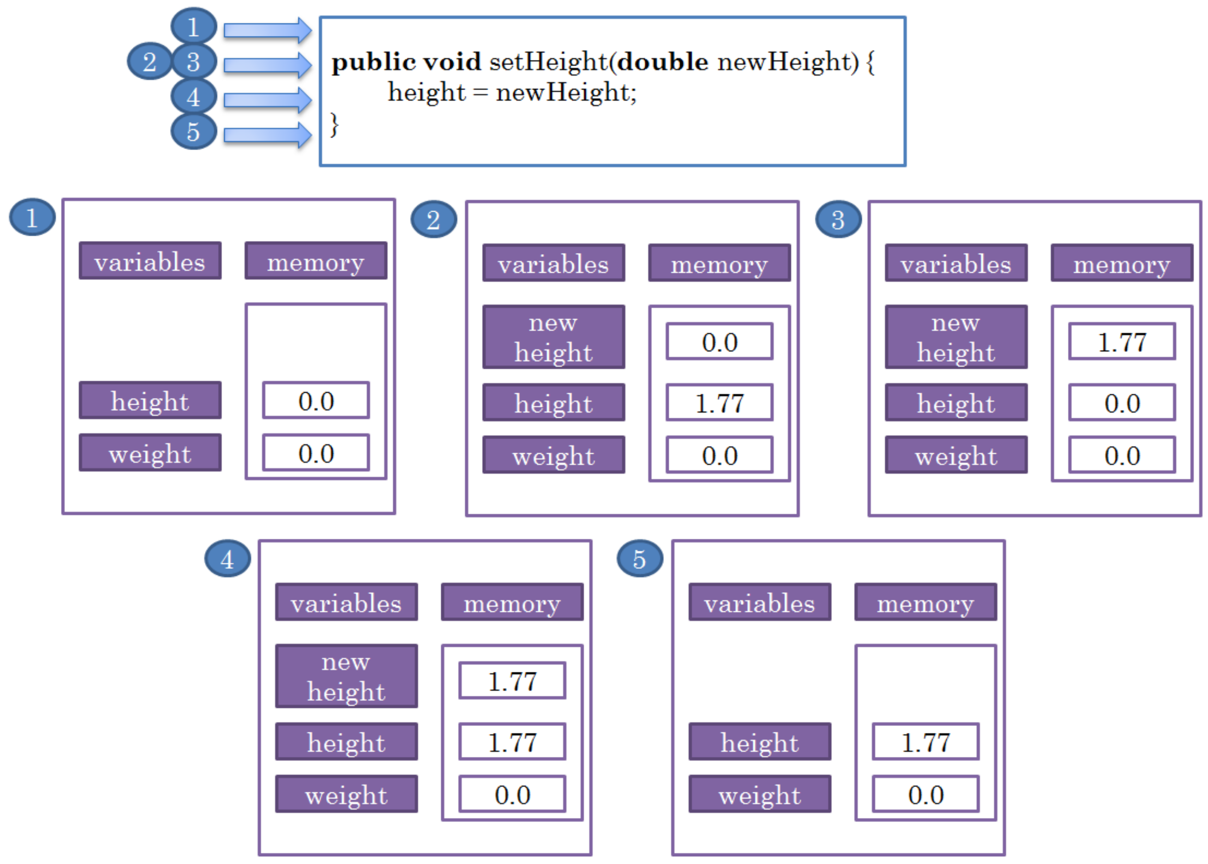


Figure 7. Events that happen when `setHeight` is called with actual parameter 1.77.
Step 1: Memory content before method executes
Step 2: Formal parameter is created when method starts
Step 3: Formal parameter is assigned actual parameter
Step 4: Instance variable is assigned formal parameter
Step 5: Formal parameter disappears when method exits

Step 4: It is now ready to execute the body of the method, which consists of a single assignment statement. The assignment statement assigns the (value of the) formal parameter, `newHeight`, to the instance variable, `height`.

Step 5: The method then exits, which results in the removal of its formal parameter from memory.

This method execution shows the importance of creating instance variables to store state that must persist between method invocations. The formal parameters of a method disappear when the method finishes execution. If their values must be used in other method invocations, they must be assigned to instance variables.

The term "bean" symbolizes a reusable component that can easily work together with other beans to create applications, much as a coffee bean can be seamlessly mixed with other coffee beans to create a coffee flavor.

Even classes that do not follow these conventions qualify as beans as long as they provide some way of specifying the getter and setter methods of properties. Naming conventions are one way of doing so. The Java beans framework provides other mechanisms to do so, which are much more complicated and meant mainly meant for classes that were written before these conventions were defined. When you are creating new classes, as in this book, it is best to simply follow these naming conventions, which have the important effect of make them easier to understand.

Properties

We now completely understand how the class `ABMISpreadsheet` works. The key ideas in this class are the concept of object state and use of getter and setter methods to export aspects of this state to classes other than the class of the object. In this example, the complete state of an instance was exported. In general, however, we may export only part of the state. For instance, we could have exported `weight` without exporting `height` had we defined `getWeight` and `setWeight` but not `getHeight` and `setHeight`. This would not be a very useful for our purposes in that it would not implement the user-interface we desire. We will later see more realistic examples in which only a part of the state of an instance is exported.

It is also possible to export a value that is not directly stored in an instance variable of an object. In our example, `getBMI` exports the BMI value, which is not stored directly in any instance variable, but can be indirectly computed from the values of the instance variables. We refer to these two kinds of properties as *stored* and *computed*, respectively.

We will refer to each unit of the state exported by a class as a *property.* More precisely, a class defines a property `P` of type `T` (Figure 8) if it declares a getter method for reading the value of the property, that is, a method with the following header:

```
public T getP()
```

If it also declares a 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*.

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 `getBMI`, as `obtainBMI`, 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 the

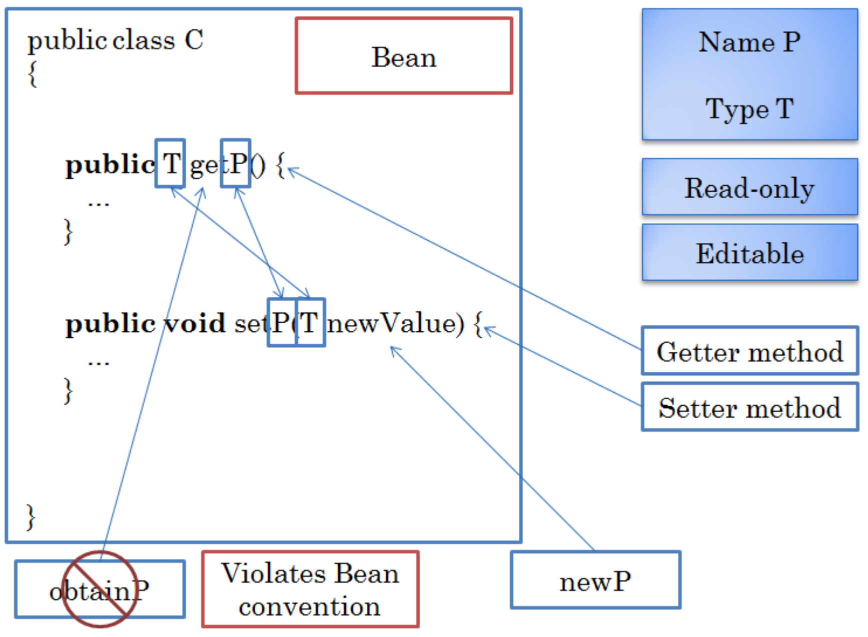


Figure 8. **A generic property declaration.**

names of the parameters of the setter methods.

On the other hands 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 such as ObjectEditor 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.

Based on above definition of properties, `ABMISpreadsheet` defines two editable properties, Weight and Height, and a read-only property, BMI. As this example shows, the properties defined by a class are related to but not the same as the instance variables of the class. The property, Weight and Height are stored in the instance variables, `weight` and `height`, but the property, BMI, is not associated with any instance variable. 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.

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 BMI spreadsheet customized for a particular adult. In such a spreadsheet, the height would never change. Thus, there would no need to make this property editable. Moreover, in such a spreadsheet, it would be useful to make the BMI property editable, and when it is set to a new value, and automatically calculate the value of weight that would result in the new BMI. Thus, in this example, height would be an independent read-only property, and weight and BMI would be editable dependent properties.

As we see in Figure 1, the edit window of an object shows all of the properties defined by its class. If a class defines no property, then the edit window is empty other than for the menus. We saw this string in the edit windows of the calculator classes, which defined no properties. If a property you defined does not appear as an item in the edit window, make sure its getter and setter methods follow the bean conventions. For instance, had we indeed named `getBMI` as `obtainBMI`, we would not see a BMI item in the edit window of Figure 1. Similarly, if `getBMI` took one or more parameters:

```
public double getBMI(double weight, double height) {
      return weight/(height*height);
}
```

it would not be a getter method.

Editable properties can be changed by users while those for read-only properties cannot. Thus, in Figure Figure 1, the text field for BMI cannot be changed by users. When we edit a text field of an editable property, a "*" (asterisk) appears next to the name of the property to indicate that field has been changed. When we next hit Enter, the following sequence of actions takes place:

1. The getter method of the property is called first! This may seem strange because, after all, we are trying to change the current value of the property, not read it. The reason the getter method is called first has to do with ObjectEditor's undo-redo mechanisms. In order to support undo-redo, ObjectEditor intercepts each change to a property and calls the property's getter method first to save the old value.
2. The setter method of the property is called with the value of the text field as the actual parameter of the method.
3. The getter methods of all properties are called and their return values used to refresh the text fields displaying their values.
4. The * next to the name of the changed property disappears to indicate that the display of the property is consistent with its actual value.

We can now understand what exactly happens when we change weight property from 77.0 to 71.0 and hit Enter (Figure 9 (top-left and top-right)).

1. The getter method, `getWeight`, is called first by ObjectEditor's undo-redo mechanism, recording the old value of 77.0.

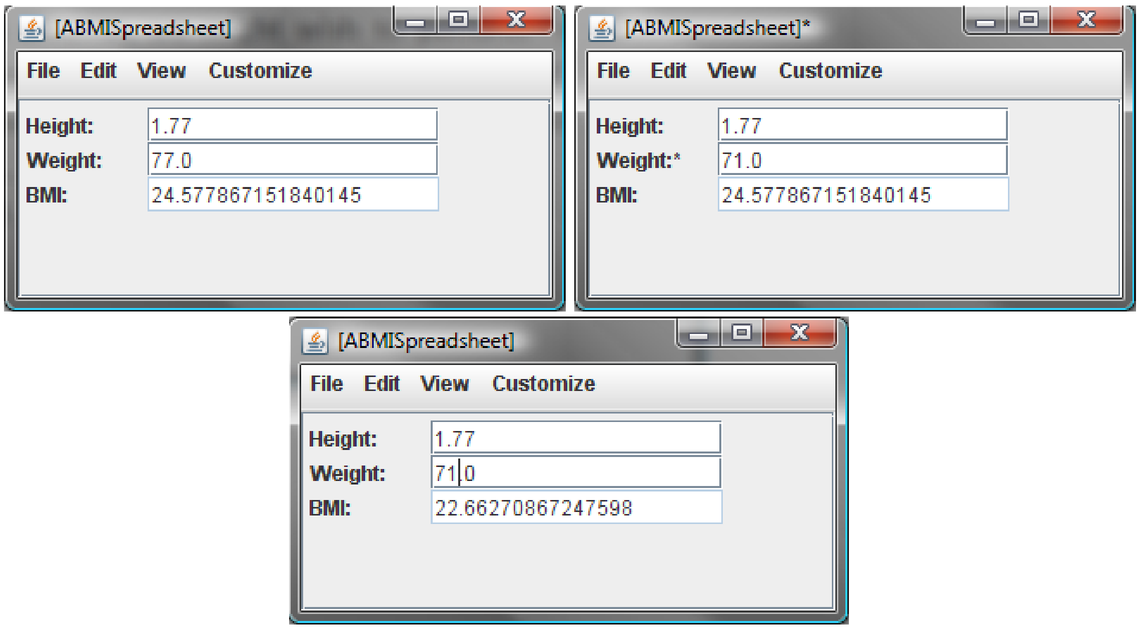
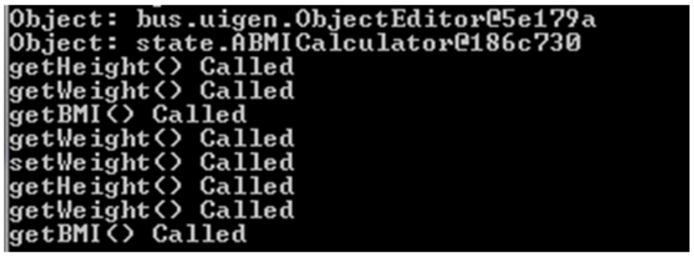


Figure 9. Setting the Weight property of `ABMISpreadsheet`



```
Object: bus.uigen.ObjectEditor@5e179a
Object: state.ABMICalculator@186c730
getHeight() Called
getWeight() Called
getBMI() Called
getWeight() Called
setWeight() Called
getHeight() Called
getWeight() Called
getBMI() Called
```

Figure 10. Tracing Execution of Getter and Setter Methods

2. The setter method, `setWeight` is called, with the parameter, 71.0. The method assigns this value to the instance variable, `weight`.
3. All of the getter methods, `getWeight`, `getHeight`, and `getBMI` are called, and their results are used to refresh the text fields displaying weight, height, and BMI, respectively. There is no change in the first two text fields, since they were displaying the current values of their properties. The text field of BMI, though, does change, because a new value of `weight` is used in the calculation Figure 9 (bottom). Ideally, ObjectEditor should have called only `getBMI`, since BMI is the only property that needs refreshing. However, it does not know the dependencies among the various properties of an object, and assumes that when one of them changes, all of them may change, and thus need to be refreshed.

The getter methods are also called when the object is first displayed in an edit window and when the refresh command is executed.

Tracing Program Execution

We can, in fact, trace the sequence of method invocations by putting special statements in the code that print text. For instance, we can insert such a statement in the beginning of the method `setWeight`:

```
public void setWeight(double newVal) {
      System.out.println("setWeight called");
      weight = newVal;
}
```

If we put similar statements in all of the methods of the class, then when we can trace the execution of the methods in the command interpreter window. Figure 10 shows the initial calls to the getter methods when the edit window of `ABMISpreadsheet` instance is first created and the calls made when the weight field is changed. As we can see, this trace is consistent with the sequence of actions mentioned above.

Statement List & Sequential Execution

As shown above, the body of the new version of `setWeight` consists of more than one statement. In general, a method body is a *statement list*, that is, a list of arbitrary statements enclosed within curly braces. When a method is called, each statement in its body is executed in order, from first to last, much as the instructions in a script are executed in order.

Statement vs. Declaration Order

While the order in which the statements appear in a statement list matters, the order in which the methods and variables[4] are declared in a class does not matter. For example, it does not matter whether the definition of `setWeight` appears before or after the declaration of `setHeight`, whether the variable `weight` is declared before or after `height`, or whether `weight` is declared before or after `getWeight` and `setWeight`. Each declaration is an independent unit. Therefore the order of declarations does not matter. On the other hand, the statements in a method body are executed sequentially in the order in which they appear. Therefore, the order in which they appear matters.

Statement vs. Expression

Beginning students often get confused between statements and expressions. Recall that we defined a statement as an instruction to the computer to perform some action and an expression as a code fragment yielding a value. The following are examples of expressions:

```
5
"setWeight called"
newHeight
```

[4] Strictly speaking, uninitialized variables. Declarations of initialized variables (which we will see later) execute statements. Therefore the order in which they appear in the program matters.

```
x*x
weight /(height * height)
```

The following are examples of statements:

```
height = newHeight;
bmi = weight /(height * height);
System.out.println("setWeight called");
return x*x;
```

The process of producing a value from an expression is called *evaluating* an expression. Depending on the expression, it may involve converting a number input by us, such as 5, to its internal representation stored in memory, looking up memory to find the value stored in a variable such as `newHeight`, invoking a function, or performing arithmetic and other predefined operations we will study later.

As we can see from these examples, an expression (in Java) is always evaluated as part of statement. For example, the expression:

```
x*x
```

is evaluated as part of the return statement:

```
return x*x;
```

and the expression:

```
weight/(height * height)
```

is evaluated as part of the assignment statement:

```
bmi = weight/(height * height);
```

A statement is an autonomous piece of code in that it does not have to be executed as part of some other statement. We have seen three different forms of statements so far: return, assignment, and print statements. A print statement is just an example of a procedure call. All procedure calls are statements, because they do not return values, and can be executed autonomously. Later, we will see other kinds of statements such as if and while statements.

Invoking Predefined Methods

Let us try to better understand the new statement we inserted in `setWeight`. It is, in fact, an example of a method invocation. We have seen before how we can invoke methods both programmatically and interactively using ObjectEditor. In this method invocation, `println` is the method, `System.out` the target object on which it is invoked, and "setWeight called" the actual parameter of the method (Figure 11). Unlike the methods we have invoked so far, this method is not defined by us - it is *predefined* by Java.

The actual parameter, “setWeight called” is an example of a *string*, that is, a sequence of characters. The double quotes indicate the start and end of the string – they are not part of it. Thus, they do not appear in the output shown in Figure 10.

println, print, & String concatenation

The effect of `println` is to display its argument in a separate line on the *console*. A console is a window created for each program to display output and error messages of the program, and as we shall see later, enter input for it. If a Java program is started from a command interpreter, then the command interpreter window serves also as the console for the program. If it is launched from an advanced programming environment, then a special console is created for it, as we shall see later.

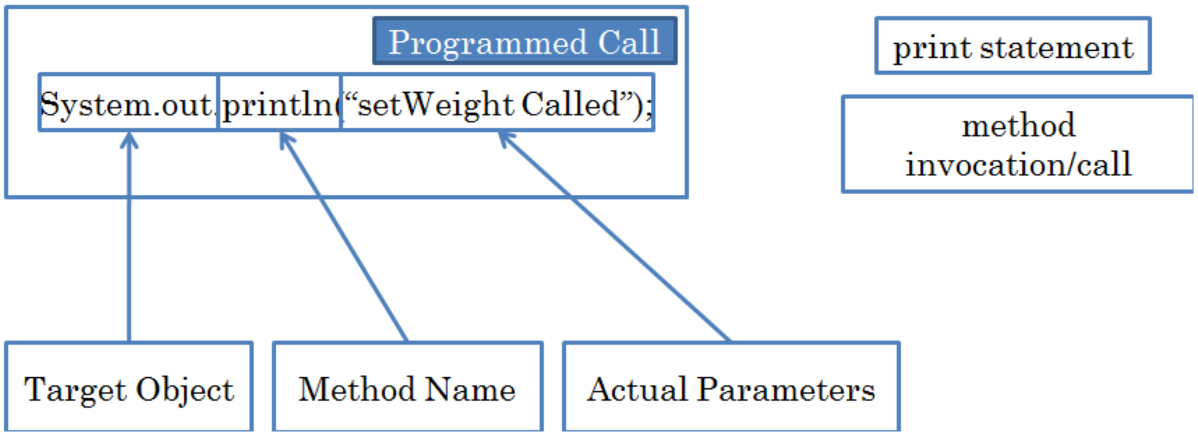
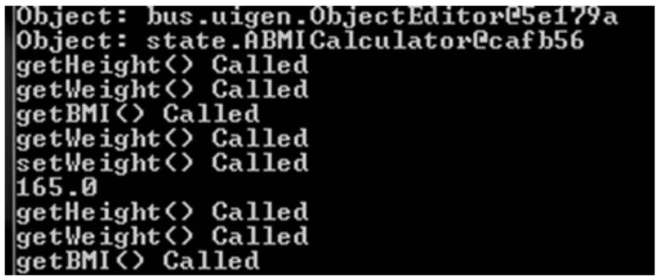
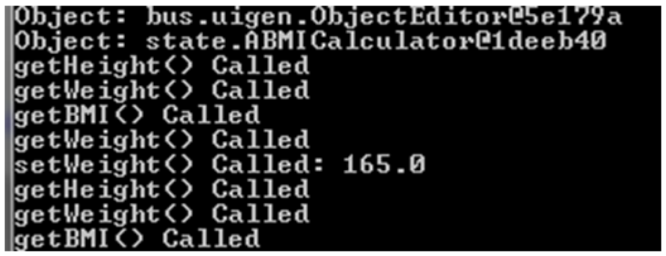


Figure 11. println as a metrhod call



```
Object: bus.uigen.ObjectEditor@5e179a
Object: state.ABMICalculator@cafb56
getHeight() Called
getWeight() Called
getBMI() Called
getWeight() Called
setWeight() Called
165.0
getHeight() Called
getWeight() Called
getBMI() Called
```

Figure 12. Printing a double



```
Object: bus.uigen.ObjectEditor@5e179a
Object: state.ABMICalculator@1deeb40
getHeight() Called
getWeight() Called
getBMI() Called
getWeight() Called
setWeight() Called: 165.0
getHeight() Called
getWeight() Called
getBMI() Called
```

Figure 13. `print` vs. `println`

The `println` method can print not just strings but any kind of Java value. If its argument is not a string, it converts the value to an appropriate string representation. For instance, we can add another statement to `setWeight` to print the value of its (**double**) parameter:

```
public void setWeight(double newWeight) {
      System.out.println("setWeight called");
      System.out.println(newWeight);
      weight = newVal;
}
```

The method will now output two lines, the second one displaying (the string representation of) the value of its parameter, as shown in Figure 12.

What if we wanted to print both strings on one line? Instead of invoking `println`, we can invoke `print`, which is like the former except that it does not create a new line after displaying its argument. Thus, we can replace the first `println` with a `print`:

```
public void setWeight(double newWeight) {
      System.out.print("setWeight called: ");
      System.out.println(newWeight);
      weight = newVal;
}
```

Figure 13 shows the new output.

In fact, there is a shorter way to achieve the same effect, using a single `println` call, as shown below:

```
public void setWeight(double newWeight) {
      System.out.println("setWeight called: " + newWeight);
      weight = newVal;
}
```

Here the argument of `println` is a concatenation or “addition” of two strings, the string “setWeight: “ and the string representation of the value of `newWeight`. Thus, the output of this method is the same as that of the version of it that uses `print`.

Concatenating multiple strings in one statement is clearly more convenient than adding multiple print statements; however, there are times when the latter approach is the only option. Suppose we wish to understand how the assignment statement in `setWeight` works by printing the value of weight before and after the statement works. Suppose also that we wish to print both values on one line. We can use the print method before the assignment and `println` after, as shown in the following code:

```
public void setWeight(double newWeight) {
      System.out.print(“weight = “ + weight);
      weight = newVal;
      System.out.println(“weight = “ + weight);
}
```

We cannot replace these two statements with a single `println`, because this statement would be executed either before or after the assignment, while we wish to know the values both before and after the statement executes.

In the rest of our discussions, we will refer to a statement that prints on the console as a print statement. In other words, we will refer to invocations of both the `print` and `println` methods as print statements.

A print statement can print any expression, not just a variable. For instance, we can print the expression computing the BMI property in `setWeight`:

```
public void setWeight(double newWeight) {
       weight = newVal;
       System.out.println(getBMI());
}
```

Recall that an expression is a code fragment that yields a value. In the print statement above, `getBMI` is called to get a double value. In general, an actual parameter of any method can be any expression that yields a value of the type of the corresponding parameter.

What we have seen above is a way to trace program execution using print statements. When we learn about debuggers, we will see a more sophisticated approach to program tracing that does not require us to first clutter the program with such statements and get rid of them later when we have finished debugging it. However, it is not always easy or possible to use a debugger – if you are using a bare-bone environment, you do not have access to a nice debugger. Moreover, you may find it difficult to learn the user-interface of the debugger. Even if you do master it, it will sometimes force you to trace statements in which you are not interested in order to see the effects of those in which you are. Thus, using print statements is a valid approach to program tracing, used even by experienced programmers.

Overloaded Operations

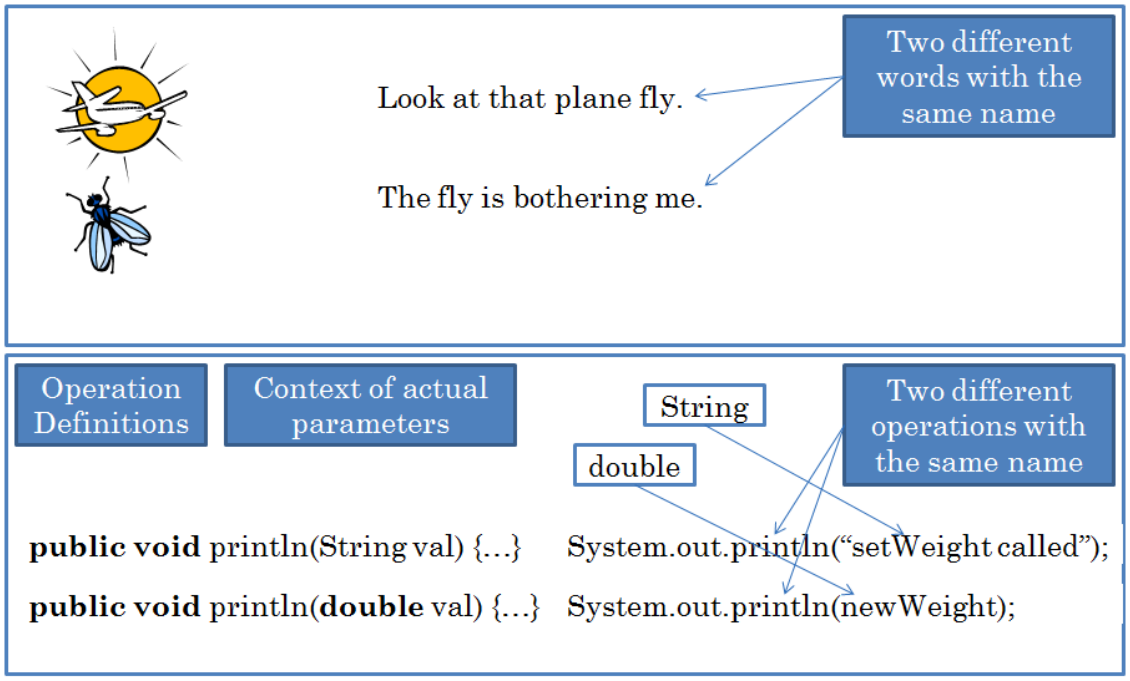
Consider the following two uses of the operator, +, in Java:

```
5 + 3
"hello" + "world"
```

The operation has different meanings in the two examples above. In the first example, it means integer addition, while the second case it means string concatenation. An operation such as + that has more than one meaning is called an *overloaded* operation. It is analogous to an English word such as "fly" that has more than one meaning. Just as we use context to determine the meaning of an overloaded English word, Java also uses context to determine the actual action performed when we use an overloaded symbol.

Like symbols, method names can also be overloaded in Java. Consider the following two uses of the method `println`:

```
System.out.println("setWeight");
System.out.println(newWeight);
```



Two different words with the same name
Look at that plane fly.
The fly is bothering me.
Operation Definitions
Context of actual parameters
String
double
Two different operations with the same name
public void println(String val) {…}
public void println(double val) {…}
System.out.println("setWeight called");
System.out.println(newWeight);

Figure 14. `print` **vs.** `println`

The second `println` is different from the one we used to print strings, since in the first case it simply prints its string argument while in the second case it converts its **double** argument to a string before printing it. As in the case of `+`, Java uses the context of the actual parameter to determine which version of `println` to call.

Suppose `System.out` provides the following two versions of `println`:

```
public void println (String val) { … }
public void println (String val) { … }
```

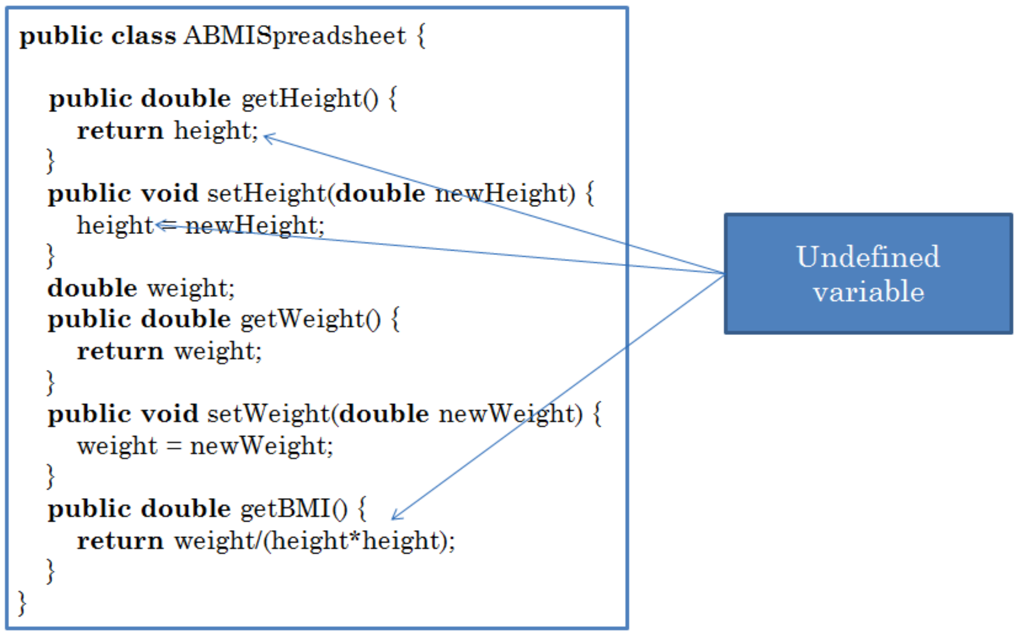
Now if we made the call:

```
System.out.println("set Weight called");
```

Java would not know which of these versions to call, because the context of actual parameters does not provide sufficient information. This is akin to the problem of parsing the following:

Time flies like an arrow. Fruit flies like an orange. (Anonymous)

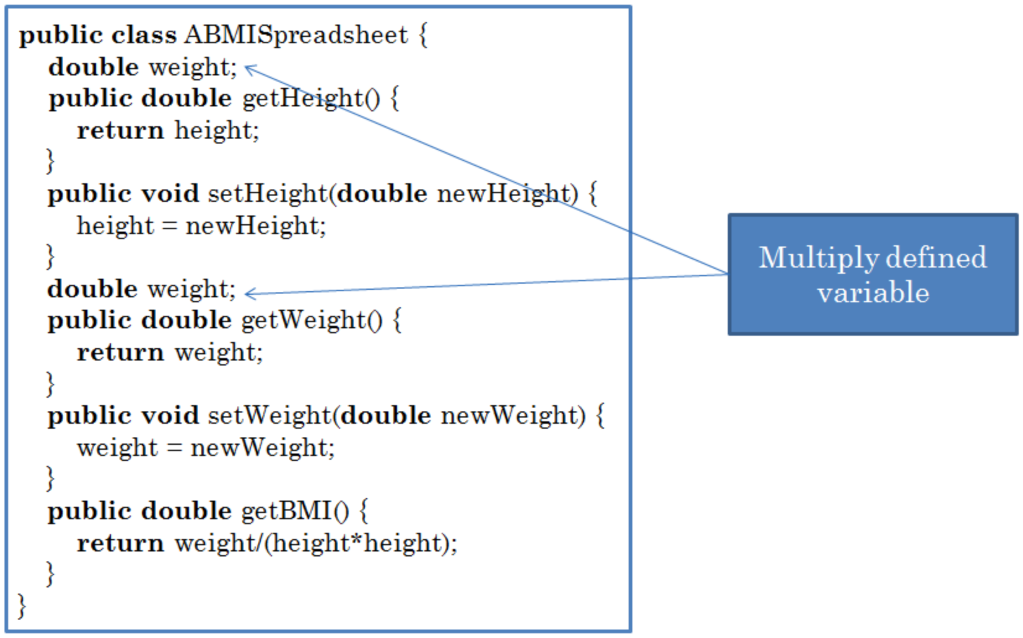
Java, therefore, forbids us to define two versions of a method in which the number of parameters are the same and corresponding parameters have the same type.



```
public class ABMISpreadsheet {

   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);
   }
}
```

Figure 15. Variable declaration error: Undefined variable



```
public class ABMISpreadsheet {
   double weight;
   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);
   }
}
```

Figure 16. Variable declaration error: multiply defined variable

Overloading allows us to use the same name for specifying similar but not identical operations. Without it, we would be forced to remember a larger number of names. For instance, to print values, without overloading, we would have to remember names such as `printlnString`, `printlnDouble`, and `printlnInt`. With overloading, Java allows us to remember a single name for all printing operations.

We will see many other examples of overloaded operations later. We will also see how we can define our own overloaded operations.

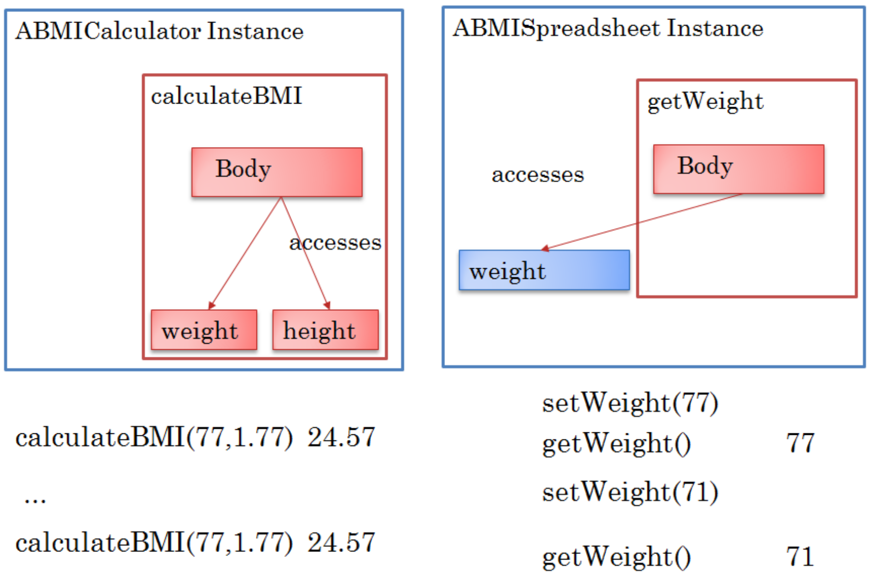


Figure 17. `print` **vs.** `println`

Undefined/Multiply-Defined Variables

Unlike in Basic and some other programming languages, in Java we must define every variable we use. Thus, we cannot omit the declaration:

```
double weight;
```

in the program above. If we do so, then at every use of the variable, Java will complain that it is not defined.

Moreover, we cannot define two variables with the same name:

```
double weight;
double weight;
```

In this case, Java will complain that the same variable has been defined multiple times.

Pure vs. Impure Functions

Let us look a bit more closely at one of the getter method above, say `getWeight`. Though it is a Java function in that it returns a result, it is not a *pure function*; that is, it is not a function in the mathematical sense of the word. The term function in Mathematics denotes a computation whose only task is to determine the relationship between its parameters and result. Two identical calls to a Mathematics function, that is, calls that provide the same actual parameters, return the same result. The function, `calculateBMI`, had this property. Thus:

```
calculateBMI (77, 1.77)
```

and

```
calculateBMI(77, 1.77)
```

return the same value: 24.57.

The method `getWeight` (and the other getter methods in this class) does not have this property. It takes no parameters, so all calls to it are identical. But what it returns depends on the current value of the instance variable, `weight`. If the variable is set to 77, then

```
getWeight()
```

returns 77; if it is set to 71, then

```
getWeight()
```

returns 71. Thus, two identical calls to the function return different values! Any function that uses values of (changeable) global variables to determine its return value is an impure function, since its arguments are not sufficient to determine its result. All getter methods in this class are impure functions.

In traditional, functional programming, impure functions were frowned upon, and often not allowed. However, such programming could not support applications such as `ABMISpreadsheet` that need state to persist between function invocations. In object-based programming, most functions tend to be impure because most real-life applications need such state.

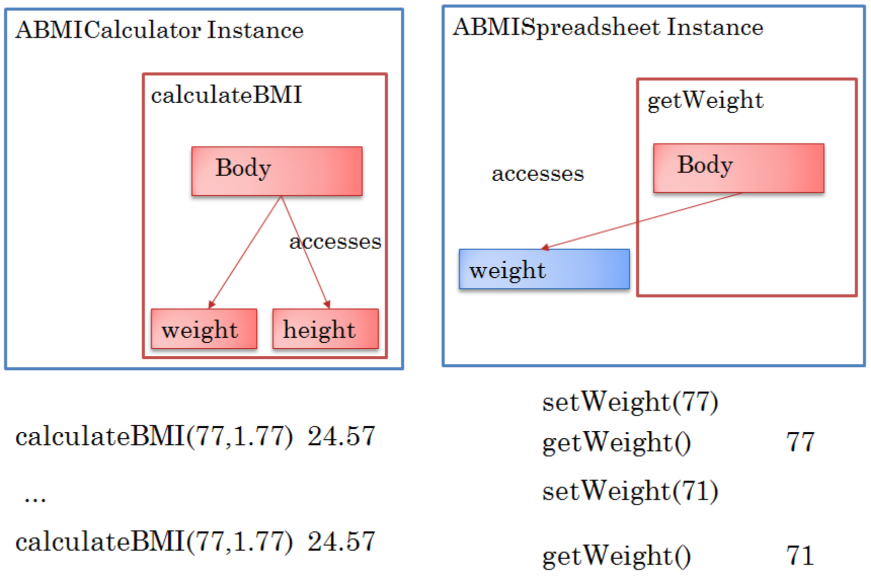


Figure 18. `print` **vs.** `println`

Side Effects in Functions

Impure functions can also have side effects. Side effects are actions performed by functions that produce output or write global variables. None of the functions we have seen see so far, not even the impure getters, have side effects. The following class illustrates side effects:

```
public class ASquareAndCubeSpreadsheet {
int number;
int square;
public void setNumber(int theNumber) {
  number = theNumber;
}
public int getNumber() {
  return number;
}
public int getSquare() {
  square = number*number;
  return square;
}
public int getCube() {
  int retVal = square*number;
  System.out.println("The Cube is: " + retVal);
  return retVal;
}}
```

Here, the following statement:

```
square = number*number;
```

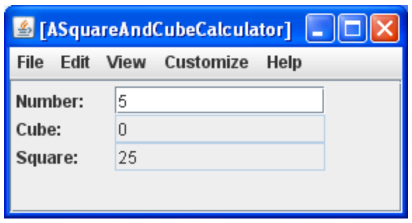
is a side effect as it occurs in a function (getSquare()) and writes a global variable (square). Similarly, the statement:

```
System.out.println("The Cube is: " + retVal);
```

is a side effect as it occurs in a function and produces (non debugging) output.

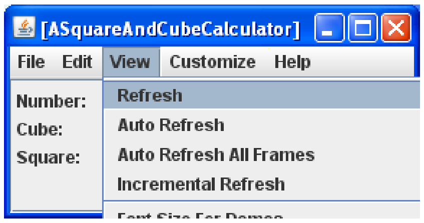
The main goal of a function is to compute its return value. To do so, it may have to read global variables, but does not have to write these variables or produce outputs. Therefore these two kinds of actions are considered side effects. Unexpected side effects make the functions that contain them difficult to understand and use correctly. There are a few situations when side effects are expected and useful – however you will not face them in this course. Therefore, avoid writing functions with side effects.

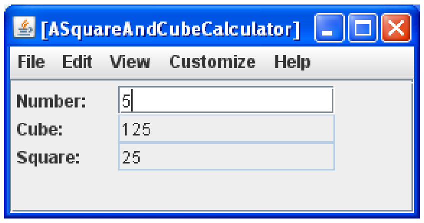
To illustrate the confusion side effects can cause, consider the following interaction with an instance fi the class above.



Number: 5
Cube: 0
Square: 25

When the user inputs a new value for Number, the Square property is computed correctly, but not the Cube property. The reason is that, after calling setNumber(5), to refresh the property values, ObjectEditor first calls getCube() and then getSquare(). Thus, when getCube() is called, the global variable, square, is still its original value, which was 0. ObjectEditor provides a command to manually refresh the display, which calls all the getters to show the current values of the properties. As we see below, if we manually refresh the window above, the Cube property is displayed correctly.





Number: 5
Cube: 125
Square: 25

The getSquare() call executed after setNumber(5) put the correct value in square. Therefore, the getCube() call executed to do the manual refresh returns the correct value.

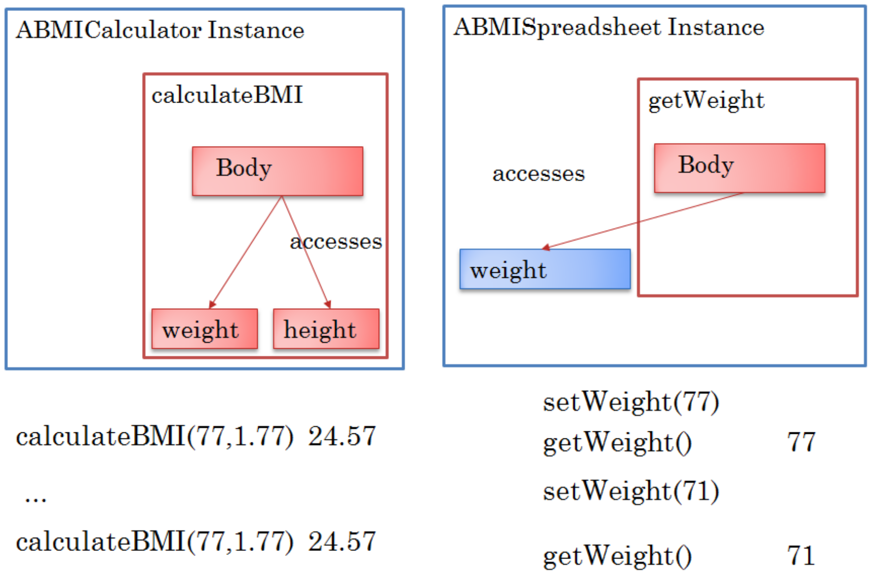


Figure 19. `print` **vs.** `println`

"Functionness" of Methods

We originally divided methods into functions and procedures based on whether they return values or not. Based on the discussion above, we can place methods in a spectrum based on how much they deviate from pure functions.

1. Pure functions: Like all functions, they compute results. In addition, they do not read or write global variables or produce output.
2. Impure functions without side effects: Like pure functions, they return results. In addition, they read global variables, but do not write global variables or produce output.
3. Impure functions with side effects: Like the functions without side effects, they return results and can read global variables. In addition, they write variables or produce output.
4. Procedures: These are like impure functions with side effects, except that they do not return results.

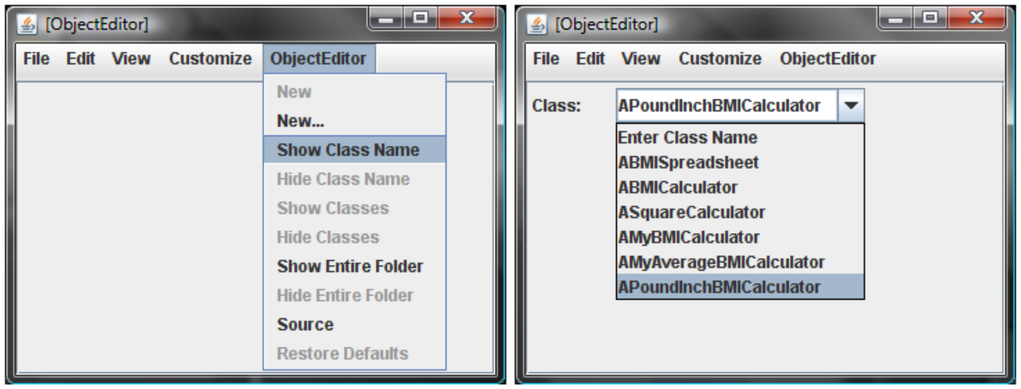
State-full ObjectEditor

The idea of state-full objects can be better understood by considering ObjectEditor. Like the BMI calculator, the user-interface of ObjectEditor we have seen so far is stateless – we have to enter the class name each time we want to instantiate a new instance. Just as we typically want to use the same value of height as we try different values of weight in the BMI case, we typically want to instantiate the same class in succession.

It is possible to transform the ObjectEditor user-interface into a state-full one, by executing the ObjectEditor→Show Class Name command (Figure 18 (left)), which shows a combobox to enter the class name (Figure 18 (right)). We can enter the name of the class we want to enter (Figure 19 (left)).

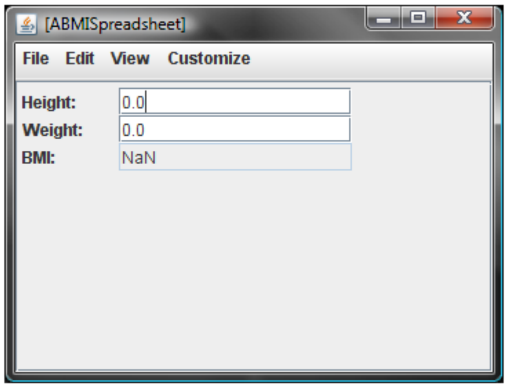
ObjectEditor remembers this value, which is now shown as a combobox choice (Figure 19 (right)). To repeatedly instantiate the same class shown as the current combobox choice, we can simply execute the parameter-less ObjectEditor→New command (Figure 20 (left)), instead of the ObjectEditor-→New… command we used earlier that took a class name as parameter.

After instantiating a class, if we wish to instantiate another one, there are two cases to consider. If we have previously instantiated it, we can simply use the combobox to change the current combobox choice. Otherwise, we can enter its name in the manner described above. Thus, once we have entered a class name, we don’t have to enter it again as we wish to create different instances of it.



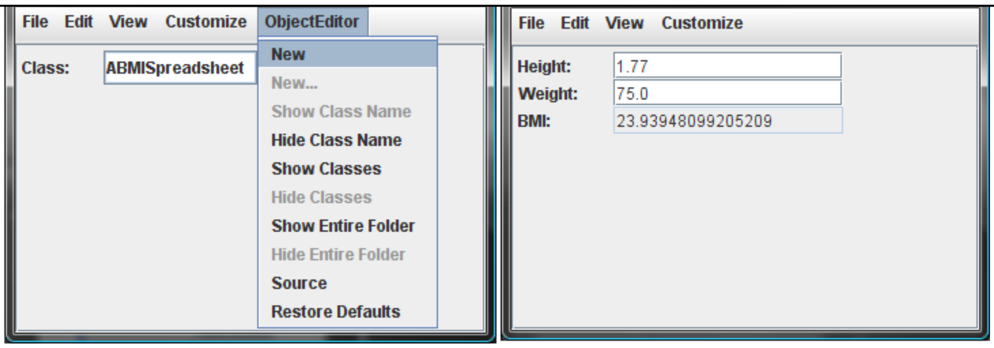
Class: APoundInchBMICalculator

Figure 23. ObjectEditor Show Class Names command reads names of classes from folder form which ObjectEditor is executed



Height: 0.0
Weight: 0.0
BMI: NaN

Figure 24. Inconsistent BMI State



Class: ABMISpreadsheet

Height: 1.77
Weight: 75.0
BMI: 23.93948099205209

Figure 22. Instantiating `ABMISpreadsheet` using the New command

Sometimes we don't need to manually enter the name of the class even the first time we instantiate it. Often we execute ObjectEditor from the folder in which we have stored classes we wish to instantiate. The ObjectEditor→Show Classes command reads the names of the classes in the folder and displays it in the combobox (Figure 21 (left)). As before, we can select a combobox item to instantiate the current class (Figure 21 (right)).

Thus we see again that the ability to remember state can result in more powerful applications.

Manipulating Properties from Code*

We have seen above how to manipulate properties interactively. The code below illustrates how this can be done from a program:

```
        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 BMI and stores this value in a local variable, and then prints the associated BMI value. Next it does an equivalent computation by calling the constructor that takes the initial values of height and weight as parameters and next printing the associated BMI value.

Objects vs. Primitives *

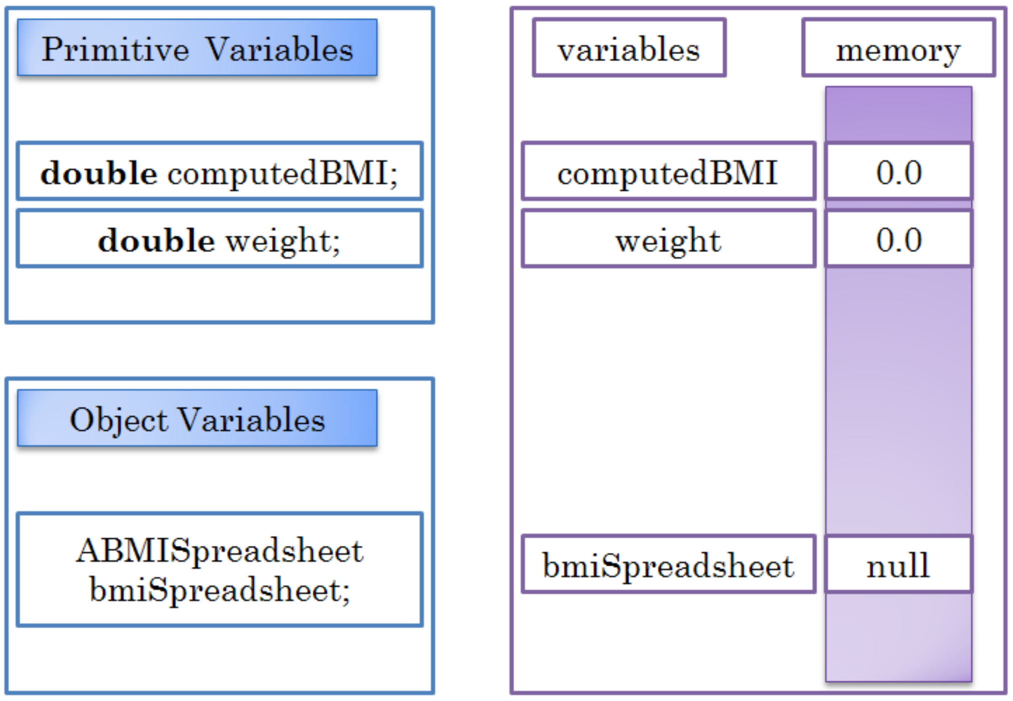
In the code above, bmiSpreadsheet is an object variable since it stores an object – an instance of the class ABMISpreadsheet. In contrast, computedBMI is a primitive variable, since it stores a double. Values of type int, char, and double are not objects. Non objects are called primitives as they are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

Default Values for Objects and Primitives *

Let us consider a variation of the code above in which neither the object or 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 bool variables it translates to the choice whose representation is 0, which is the **false** value. For object variables, it translates to a special object value called **null**. The following figure illustrates default values of object and primitive variables.



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, Java gives us a NullPointerException.

Constructors

A problem with state-full objects is that instance variables can have illegal values. Consider the object we get when we instantiate `ABMISpreadsheet`, shown in Figure 22. 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, 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 initial values of the height and weight and initializes their values 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;
}
```

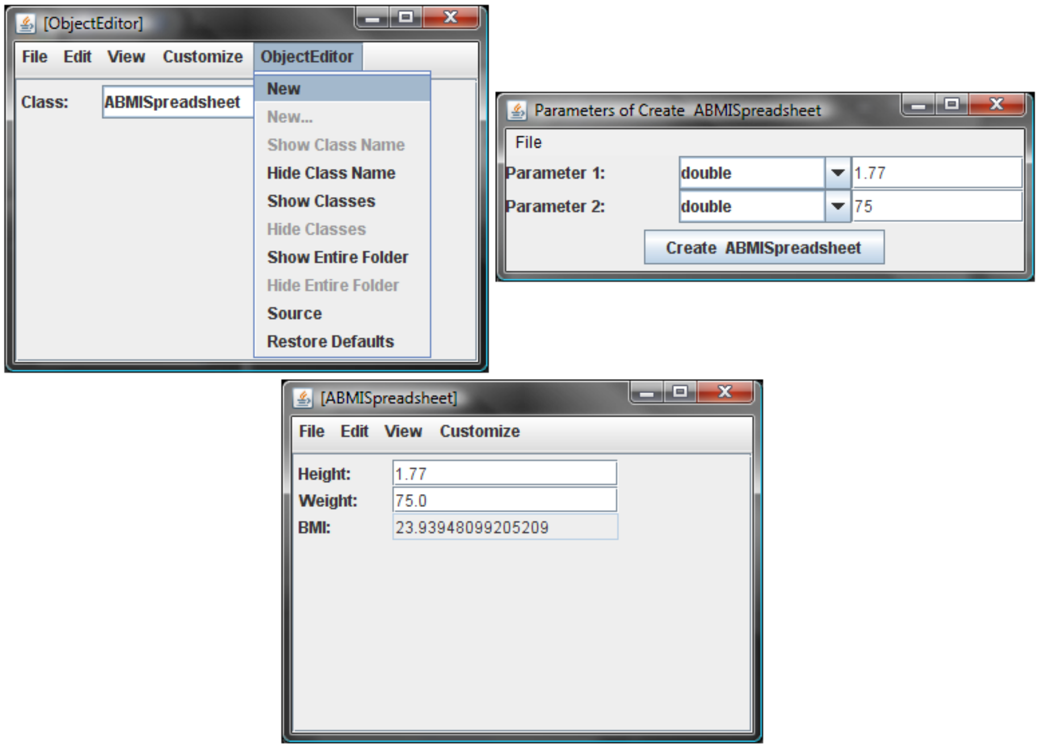


Figure 25. Interactively instantiating `ABMISpreadsheet` with a consistent state: (top-left) Invoking New, (top-right) filling the constructor parameters, and (bottom) the initialized instance

Like the `new` expression we write in a Java program, the New command provided by ObjectEditor also calls the constructor of the class to be instantiated (Figure 23 (top-left)). If the constructor takes a parameter, ObjectEditor prompts the user for values of the actual parameters (Figure 23 (top-right)). Thus, if we now try to instantiate `ABMISpreadsheet`, we will be asked to fill the initial values of height and weight before the object is created. After entering the value, we can click on the Create ABMISpreadsheet button to create and edit the new instance. The edit window created now reflects the values entered as the parameter (Figure 23 (bottom)).

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.

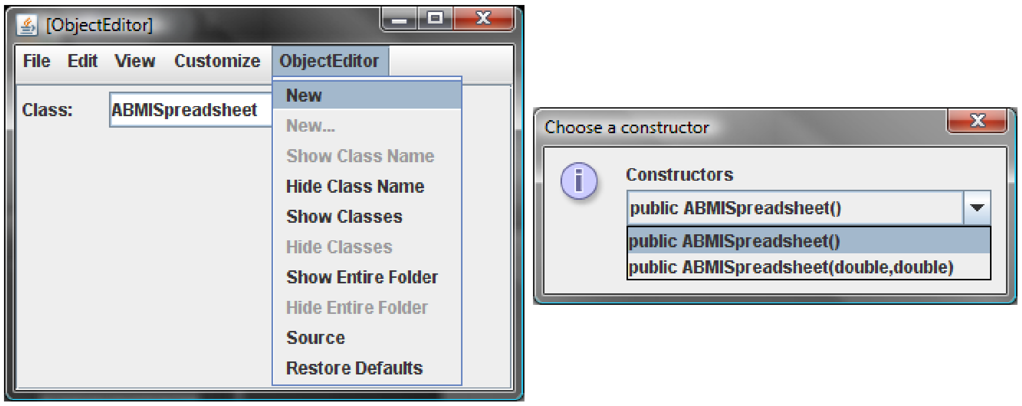


Figure 26. Interactive illustration of overloaded constructors

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:

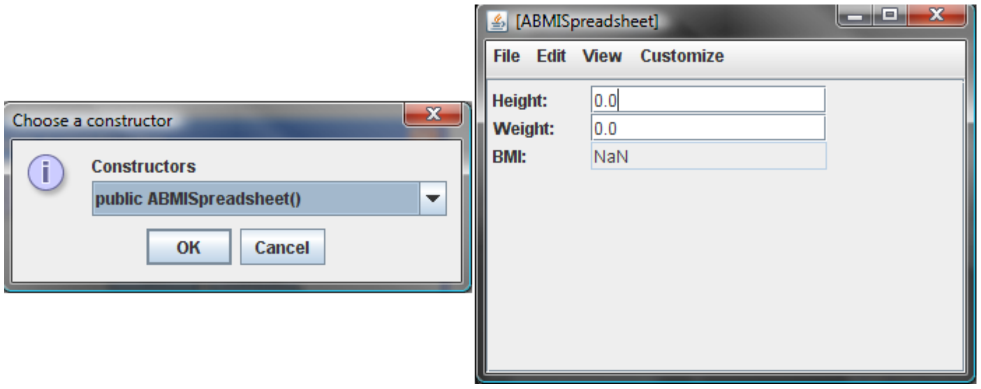
```
public ABMISpreadsheet ABMISpreadsheet(
            double theInitialHeight, double theInitialWeight) {
      height = theInitialHeight;
      weight = theInitialWeight;
}

public ABMISpreadsheet() {}
```

Thus, like ordinary methods, constructors can be overloaded.

The interaction with ObjectEditor shown in Figure 24 graphically illustrates the concept of overloaded constructors.

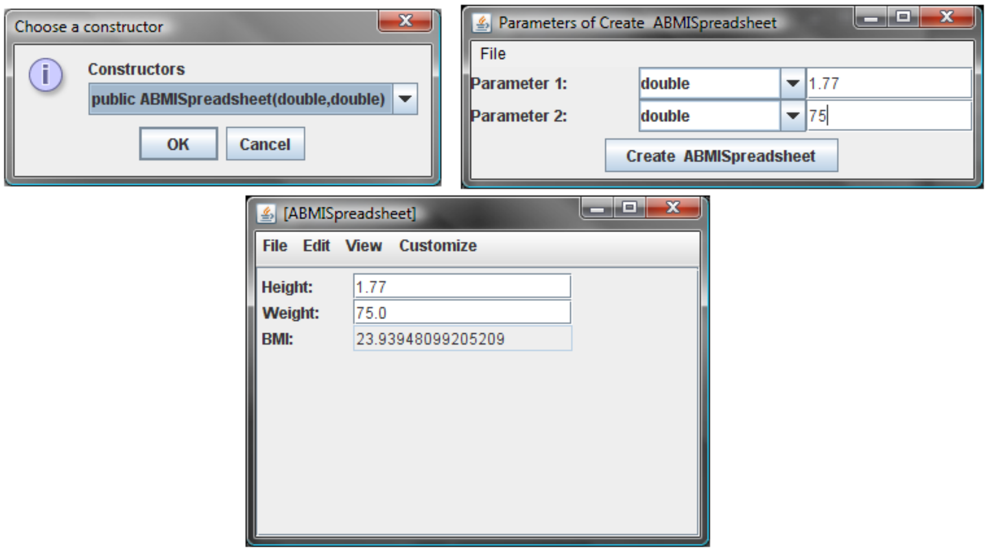
When we execute the ObjectEditor→New command to instantiate the version of the ABMISpreadsheet class with two constructors, ObjectEditor shows a combobox displaying the headers of both of these constructors. By choosing the appropriate combobox item, we can invoke the desired constructor. Figure 25 and Figure 26 show what happens when we take each of the two choices.



Constructors: public ABMISpreadsheet()

Height: 0.0
Weight: 0.0
BMI: NaN

Figure 27. Creating an instance of `ABMISpreadsheet` **using the parameter-less constructor**



Constructors: public ABMISpreadsheet(double,double)

Parameter 1: double 1.77
Parameter 2: double 75

Height: 1.77
Weight: 75.0
BMI: 23.93948099205209

Figure 28. Creating an instance of `ABMISpreadsheet` using a constructor with parameters

Invoking Constructors from Code*

We have seen above how to invoke constructors interactively. Later, when we discuss structured objects, we will consider in depth how we can invoke constructors from code we write. The code below illustrates how this is done:

```
public class BMISpreadsheetUser {
    public static void main(String[] args) {
        ABMISpreadsheet bmiSpreadsheet = new ABMISpreadsheet(1.77, 75);
        double computedBMI = bmiSpreadsheet.getBMI();
        System.out.println(computedBMI );
    }
}
```

This code is much like the code we saw earlier for instantiating ABMISpreadsheet, except that it calls the two parameter constructor instead of the default constructor. Thus, there is no need to call the setter methods to initialize the object.

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

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.

Summary

- State that must persist between invocations of the methods of a class can be stored in instance variables of the class, which are global, that is, accessible from all methods declared in the class.
- Part of this state can be exported to other classes as editable and read-only properties, which are defined by getter and setter methods.
- A getter method returns the value of a property, and a setter method sets the value of an editable property.
- A method can be classified as a function or a procedure based on whether or not it returns a value. Getter methods are functions and setter methods are procedures.
- Getter methods are impure functions whose results depend on values other than their local variables.
- An assignment statement can be used to store the expression on its RHS in the variable on its LHS.
- A method body can consist of a series of statements, each of which is executed in sequence.
- Print statements provide a general way to trace program execution that is not tied to a specific programming environment.
- Overloaded operations such as a print method or the + operator are associated with multiple implementations; which one is used depends on the context.

Exercises

1. Distinguish between functions and procedures, pure and impure functions, and dependent and independent properties.
2. Consider the following code for representing a certain amount of money. The first two methods set and get the amount in dollars. The third returns the amount in cents. The fourth computes an amount in cents from an amount in dollars.

```
public class C () {
      public int m = 100;
      public int getD() {
                  return m;
      }
      public void setD(int i) {
            m = i;
      }
```

```
        public int getC() {
              return m*100;
        }
        public int getCInD (int i) {
              return i*100;
        }
  }
```

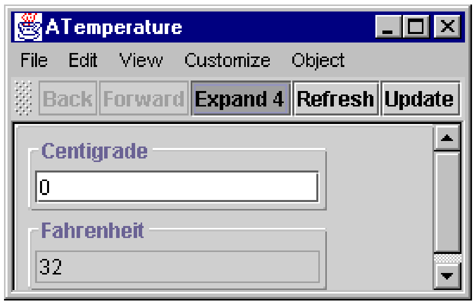
a) Identify the properties defined by this class, classifying them into read-only, editable, dependent, and independent.

b) Classify the methods of this class into procedures and functions and the functions into pure and impure functions.

c) Classify the variables declared in this class into instance variables and formal parameters.

Rewrite the code using better names for the class, methods, and variables.

3. Create a spreadsheet version of the temperature converter class from Chapter 1, Exercise 6. The new class should define an editable property specifying the value of a temperature in Centigrade and a read-only property specifying its value in Fahrenheit, as shown below:


ATemperature
File Edit View Customize Object
Back Forward Expand 4 Refresh Update

Centigrade
0
Fahrenheit
32

Print the value of each instance variable of the above class before and after it is assigned a new value.