COMP 110

Prasun Dewan[1]

3. Functions

The (classes of the) objects we saw in the previous chapter are rather simple. Not because they are small in size. In fact, in well-written, real-world code, objects tend to be fairly small. The main reason for the simplicity is that each of these objects has a single, self-contained method that does not interact with other methods. In this chapter we will see how methods can call each other, much like we humans delegate work to others. Such collaborative methods do not increase the complexity of the kind of applications we build. Instead they improve the style of the program, making it easier to understand, change, debug, and get correct. We will see some additional elements of programming style such as named constants and comments. Writing complex collaborative methods often requires to break the application development process into phases in which we first develop the high-level steps that decide what the computer should do and phases in which we translate these steps into code in some programming language such as Java. We will look also at this stepwise refinement process.

A Personalized BMI Calculator

A BMI calculator lets us determine how much weight we should lose or gain to be in the range of acceptable BMIs. However, the BMI calculator we coded in the last chapter is a general purpose one that can be used by people of different heights. As a result, each time we want to try a new value of our

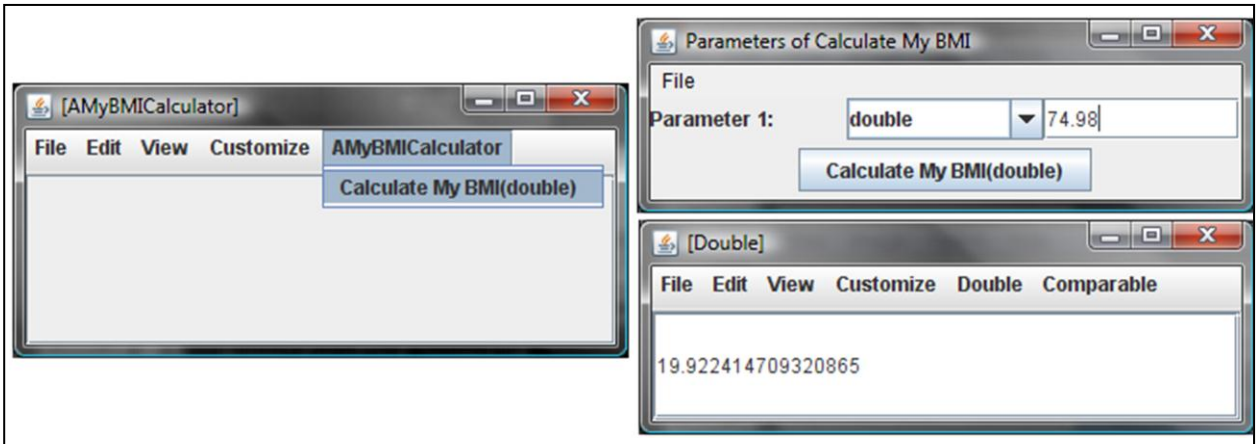

Figure 1. Using `AMyBMICalculator`: (left) Method `CalculateMyBMI` of class `AMyBMICalculator`, (top-right) the method only requires weight, and (top-left) the calculated BMI value

```
public class AMyBMICalculator {
   public double calculateMyBMI(double weight) {
      return weight/ (1.94 * 1.94);
   }
}
```

Figure 2. The class `ASquareCalculator`

weight, we must also enter our height. The BMI calculator shown in Figure 1, called `AMyBMICalculator`, is a custom one built for a particular individual. Thus, we can simply try different values of weight, without entering our height each time.

Figure 2 shows the code for the new class. As expected, the method, `calculateMyBMI` of this class takes a single parameter – the weight. The height, 1.94, is hardwired into the method. It uses the fixed height to compute the BMI corresponding to the weight entered by its caller. Thus, there is very little difference between this method and the `calculateBMI` method of the previous chapter except in whether the height is hardwired or a parameter. In fact, we can simply copy the code of `calculateBMI` from the previous class, paste it into this class, and make the minor edits of removing the height parameter from the method header and replacing its occurrences in the method body with the known value of height.

Copy-Paste-Edit vs. True Reuse

As we program, we are often tempted to "reuse" code by going through the copy-edit-paste process used above. However, this process has the following problem. It duplicates code, which means we must find and modify all copies of the code if we must change its behavior or fix errors. Suppose we had made a mistake in coding `calculateBMI` by, for instance, using the `+` operator instead of the `*` operator:

```
public double calculateBMI (double weight, double height) {
      return weight/ (height + height).
}
```

After the copy-paste-edit process, our `calculateMyBMI` method looks like:

```
public double calculateMyBMI (double weight) {
      return weight/ (1.94 + 1.94).
}
```

We must now change both methods to correct the mistake. This mistake is fairly easy to identify and fix. In general, as you know from using commercial applications, there are many mistakes that are hard to find even by experienced programmers. The copy-paste-edit process compounds the problem of fixing mistakes as the effort required to find the mistake in copied code might be duplicated if we do not keep track of where the code has been copied. Fortunately, there are elegant ways to address this problem that supports *true reuse* by keeping a single copy of code that must be executed in different parts of the

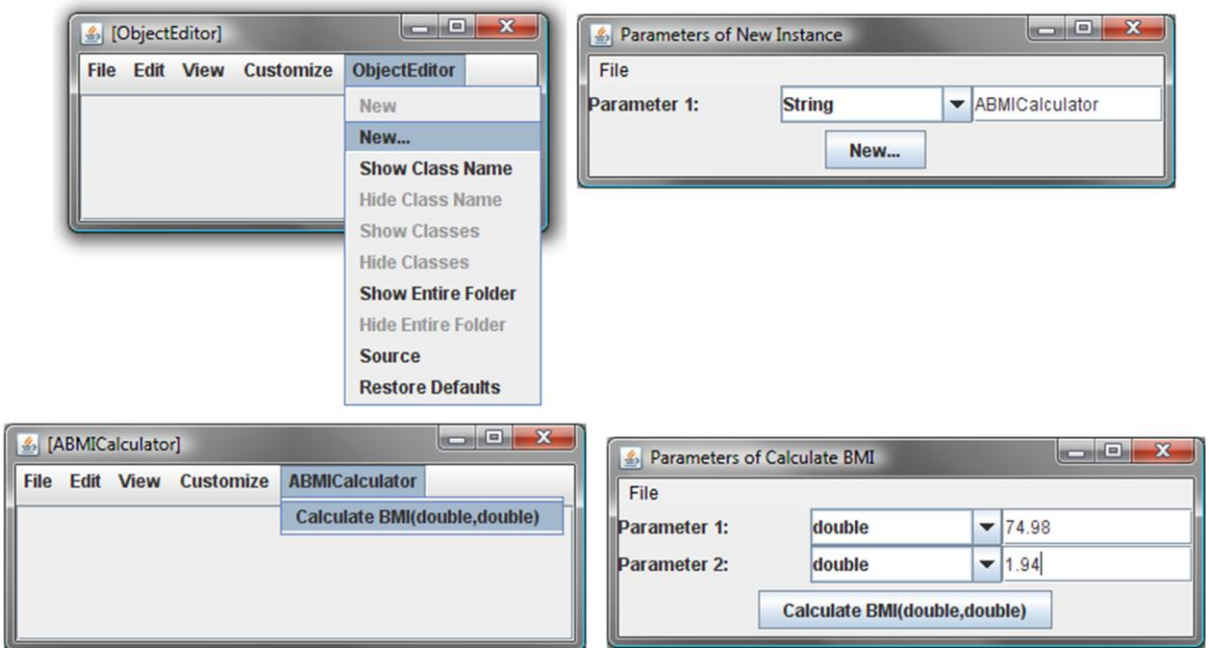

Figure 4. Calling `calculateBMI` interactively

application. We will study here the simplest of these techniques, which involves calling rather than duplicating method code.

Calling Methods Programmatically

We can take the following steps in `calculateMyBMI` to truly reuse the code in `calculateBMI`:

1) Create an instance of `ABMICalculator`
2) Invoke the method `calculateBMI` on this instance passing it my weight and my height as actual parameters
3) Return the value returned by `calculateBMI`

When we coded `calculateBMI`, we learnt how to return a value from a method. Thus, we know how to perform the third step. Moreover, we have seen how to execute the first two steps interactively. Figure 4 summarizes them. Java provides us with equivalent ways to perform these steps from the program, which are shown in the modified `calculateMyBMI` method shown in Figure 3.

```
public class AMyBMICalculator {
   public double calculateMyBMI(double weight) {
      return (new ABMICalculator()).calculateBMI(weight,1.94);
   }
}
```

Figure 3. The class `AMyBMICalculator` that reuses `ABMICalculator`

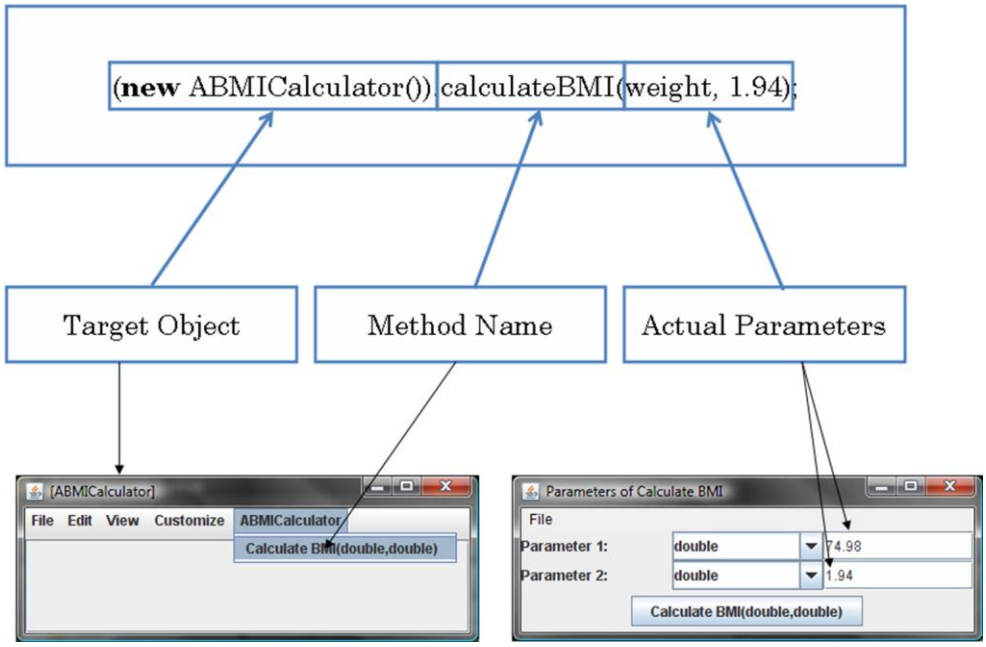

Figure 5. Programmatically instantiating and using `ABMICalculator`

The following fragment illustrates how a class is instantiated:

```
new ABMICalculator()
```

Why the parentheses after the class name? The reason will be clear later when we study constructors. The parentheses are for enclosing a list of actual parameters given when instantiating a class. This implies that `ABMICalculator` above is not only the name of the class to be instantiated but also the name of a method that processes the actual parameters.

In general, if `C` is a class, the expression:

```
new C(…)
```

returns a new instance of the class, where the … are actual parameters given when instantiating the class.

Thus, programmatic and interactive instantiation of a class are very similar. This is also true for method invocation. We invoke a method interactively by selecting the method name from a menu in the edit window of the "target" object on which the method is to be called, and filling the values of the parameters of the method. The labeled program fragment in Figure 5 how these steps are taken in a program.

In general, the syntax for invoking method `M` on object `O` with parameters `(p1, p2, …)` is:

```
O.M(p1, p2, …);
```

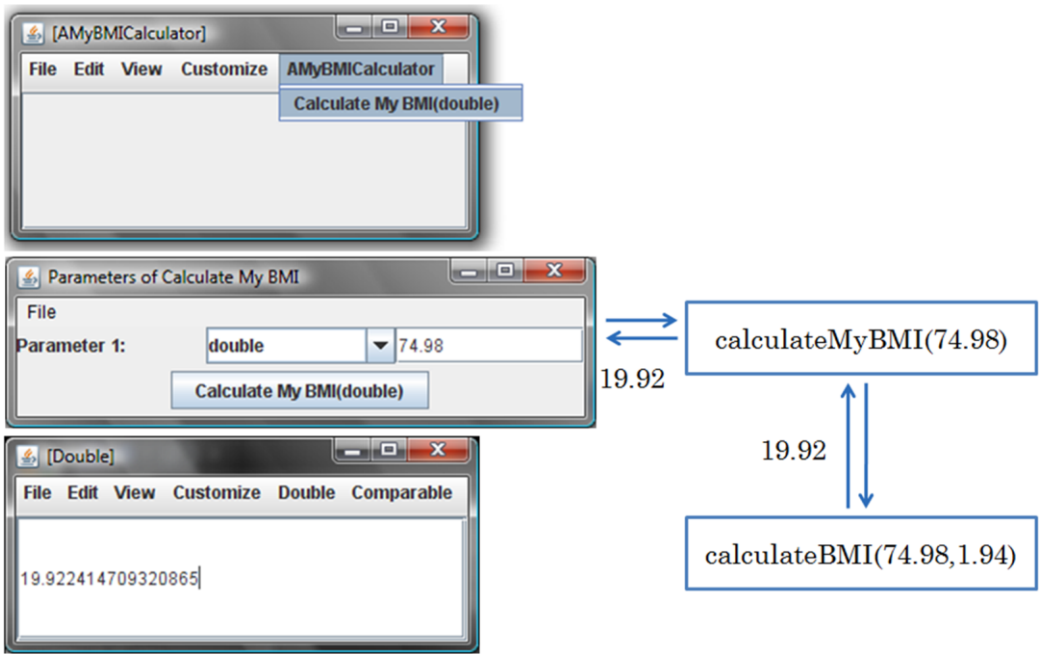

Figure 6. Passing the buck by calling a method

Thus, `calculateMyBMI` creates an instance of `ABMICalculator`, invokes the method `calculateBMI` on this target object with the known value of height and the value stored in the formal parameter `weight`, and returns the value returned by `calculateBMI`. Thus, `calculateMyBMI` does not do any "real work" – it simply passes the buck to `calculateBMI`. This is good as we have true reuse. We have not duplicated the formula for calculating the BMI value in different parts of the program. If we fix any errors in it or change it, for instance to take height and weight in feet and inches, respectively, we have to change only one piece of code. In general the calling and called methods (`calculateMyBMI` and `calculateBMI`, in this example) are referred to as the *caller* and *callee*, respectively. Figure 6 graphically illustrates the notion of passing the buck by calling a method.

When the user enters 74.98 for the weight parameter and presses the button to invoke the method, `calculateMyBMI`, ObjectEditor calls `calculateMyBMI(74.98)`, which in turn calls, `calculateBMI(74.98, 1.94)`, using the known value of height. The later method returns 19.9 to `calculateMyBMI`, which in turn, returns this value to ObjectEditor, then displays the value to the user.

Declaring Variables in a Method Body

Consider another variation of the personalized BMI calculator shown in Figure 7. It defines a method that takes as parameters two weights, such as our current weight and the weight a year ago, and finds the average of the BMIs corresponding to the two weights.

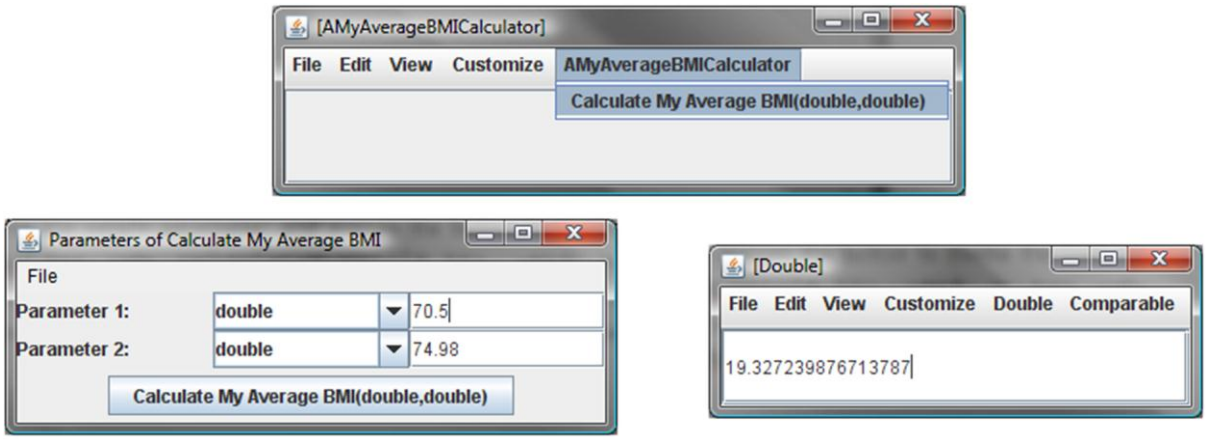

Figure 7. Interactively instantiating and using `AMyAverageBMICalculator`

```
public class AMyAverageBMICalculator {
   public double calculateMyAverageBMI(double weight1,
                                       double weight2) {
      return ((new ABMICalculator()).calculateBMI(weight1, 1.94) +
         (new ABMICalculator()).calculateBMI(weight2, 1.94))/2;
   }
}
```

Figure 8. Programmatically creating two instances of `AMyAverageBMICalculator` **to calculate average BMI**

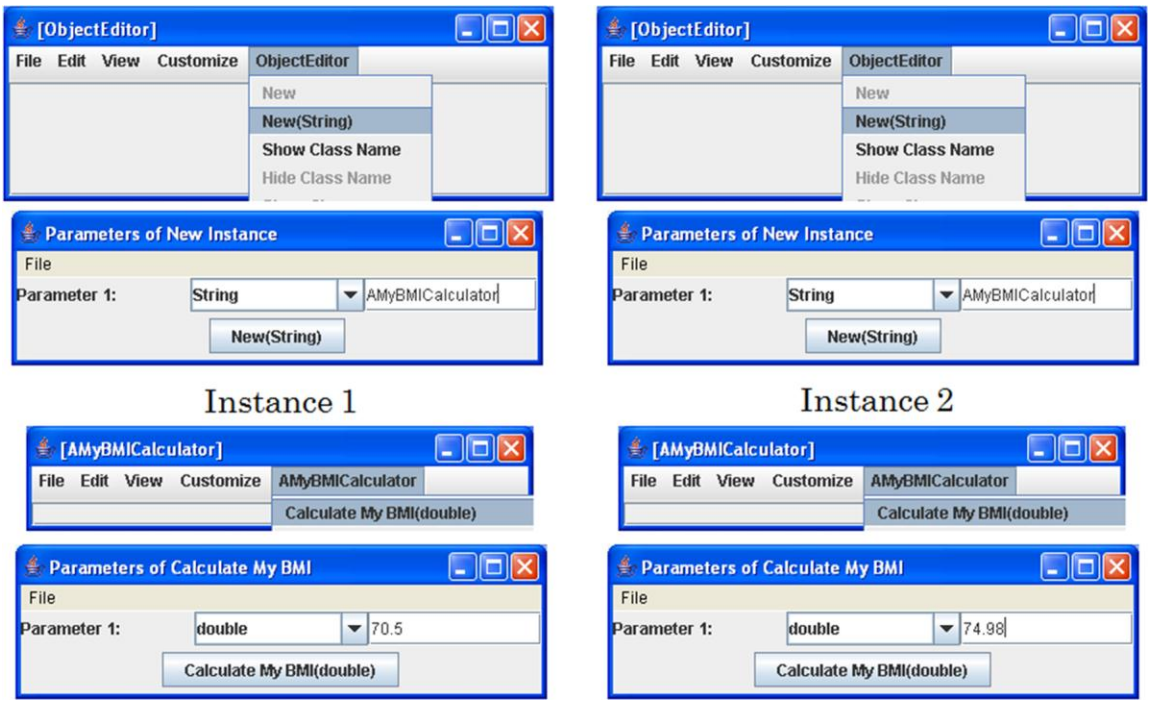

Figure 9. The interactive equivalent of programmatically creating two instances of `AMyAverageBMICalculator` **to calculate average BMI**

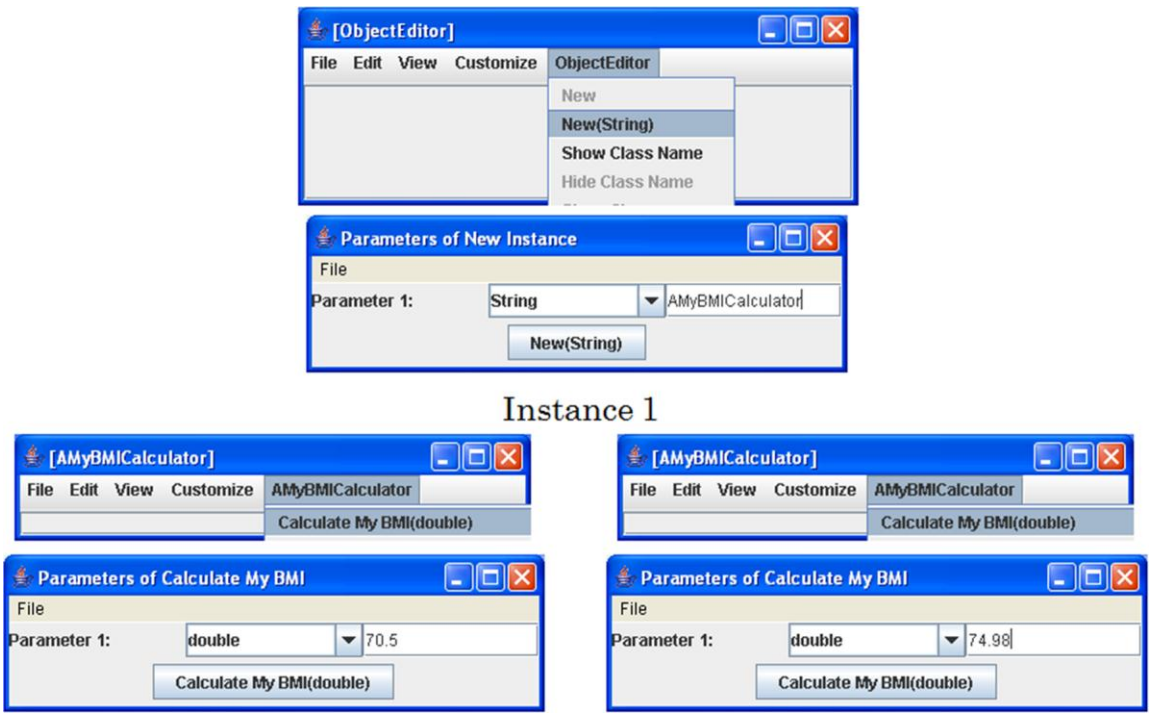

Figure 10. Interactively creating a single instance of `AMyBMICalculator` and using it twice

Without knowing additional Java constructs, we would have to implement this class as shown in Figure 8. However, the method, `calculateMyAverageBMI` creates a new instance of `ABMICalculator` each of the two times it wishes to invoke the method `calculateBMI`. This corresponds to the interactive execution of the two methods as shown in Figure 9.

Here the labels, "Instance 1" and "Instance 2", identify the windows ObjectEditor creates for the two instances of `AMyBMICalculator` created by the user. Of course, we did not have to do so. Instead the interaction shown in Figure 10 is much less tedious.

In fact, it is normal to invoke multiple methods on an object after it has been created. To do so, we need to a way to identify the object on which we wish to invoke the method. In the interactive mode, the window ObjectEditor creates for an instance identifies it. In a program, we can use variables to do so. Each program value such as an object or a double is stored in some location in memory. We can use variables to name some of these locations in a program. Before using a variable we must first declare its name and the type of the object stored in the corresponding memory location.

In fact, we have already seen the concept of declaring and using a variable. Consider again the code shown in Figure 11. The method header contains the following variable declaration:

```
double weight
```

The body of the method references the variable:

```
return weight/(1.94 * 1.94);
```

```
public class AMyBMICalculator {
   public double calculateMyBMI(double weight) {
      return weight/ (1.94 * 1.94);
   }
}
```

Figure 11. Original `AMyBMICalculator` **code**

Each reference of the variable stands for the value stored in the corresponding memory location. Thus, if the actual parameter, 74.5 is passed to `calculateMyBMI`, then the above code is equivalent to:

```
return 74.5/(1.94 * 1.94);
```

As we will see later, the initial value of a variable can be changed, hence the name variable.

Variables can be declared not only in the header of a method as formal parameters but also its body as *internal method variables*. While a formal parameter has a well defined initial value, the actual parameter, an internal method variable does not. Therefore, Java lets us explicitly define such a value when we declare the variable. The alternative implementation of `calculateMyBMI` shown in Figure 12 illustrates this.

Here, we create the variable names, `aBMICalculator`, in the body of the method `calculateMyBMI`.

```
public class AMyAverageBMICalculator {
   public double calculateMyAverageBMI(double weight1, double weight2) {
      ABMICalculator aBMICalculator = new ABMICalculator();
      return (aBMICalculator .calculateBMI(weight1, 1.94) +
            aBMICalculator.calculateBMI(weight2, 1.94))/2;
   }
}
```

Figure 12. `AMyBMICalculator` **code with storing** `ABMICalculator` **as an internal method variable**

```
public class AMyAverageBMICalculator {
   public double calculateMyAverageBMI(double weight1,
                                       double weight2) {
      ABMICalculator aBMICalculator = new ABMICalculator();
      double bmi1 = aBMICalculator.calculateBMI(weight1, 1.94);
      double bmi2 = aBMICalculator.calculateBMI(weight2, 1.94);
      return (bmi1 + bmi2)/2;
   }
}
```

Figure 13. `AMyBMICalculator` **code with multiple internal method variables**

Its type, `ABMICalculator`, indicates that its value is an instance of `ABMICalculator` .The variable is initialized to a new instance of this class.

In general, the construct:

```
T V = E;
```

asks Java to declare a variable named `V` of type `T` and stores the value `E` in its memory location. The variable, `V`, now stands for the value, `E` (until it is dynamically assigned a new value). Thus, the above program calls the method `calculateBMI` on the instance stored in `aBMICalculator` twice, with different values of the `weight` parameter. Compared to the previous implementation, this one is more efficient as we created one rather than two objects for the two method invocations.

In fact, it is possible to use additional variables in the code as shown in Figure 13. In Figure 13, after each invocation of the `calculateBMI` method on `aBMICalculator`, the return value is stored in a variable. Doing so is not necessary and does not increase the efficiency of the code. However, it may make the code clearer to some of you. More importantly, as we see later, it is easier to debug, as we can name all the intermediate values computed by the program. On the other hand, it makes the code less concise. Which implementation is used is, thus, a matter of taste.

Named Constants, Literals, Constants

Nevertheless, there are arguably good ways to improve programming style, just as there are arguably good ways to improve writing style. Consider the implementation:

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

Here the value 1.77 may be a *magic number* to some readers of the code, that is, a mysterious number whose role they do not understand. The implementation would make more sense if we rewrite our code as:

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

Here MY_HEIGHT is a *named constant*. A named constant is like a variable in that it has a name (identifier) associated with a value. While the association between a variable and its value can change, that is, a variable can be assigned different values, the association between a named constant and its value does not change. This is the reason for putting the keyword **final** in the declaration of the variable. The convention is to capitalize all letters of a named constant to distinguish it from a regular variable and separate its various words with an "_" (e.g. `MY_HEIGHT`).

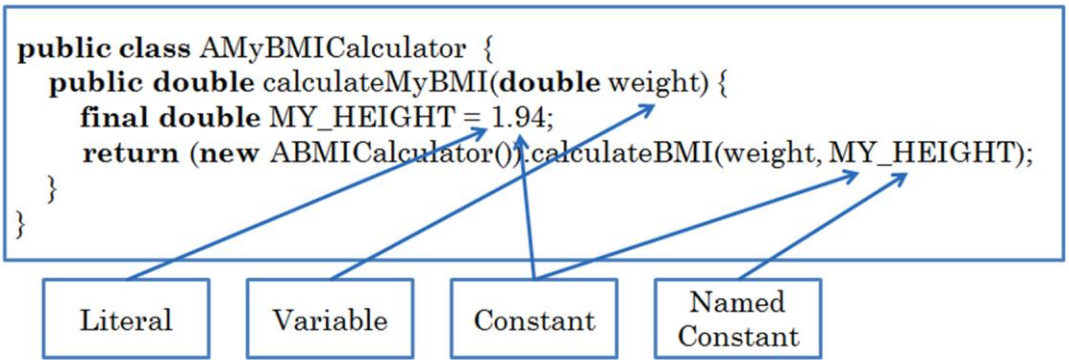

Figure 14. Literals, constants, named constants, and variables

Both the number 1.77 and the named constant `MY_HEIGHT` are constants, that is, their values are fixed. The former is called a *literal* to distinguish it from the latter. A literal is a constant that literally indicates the value it represents. A named constant is not a literal because we must look up its declaration to determine its value. Literals are not restricted to numbers such as 2.2. As we will see later, they may be characters such as '`h`', strings such as "`hello`," or **boolean** values such as `true`.

Figure 14 illustrates the difference between literals, named constants, constants, and variables.

You might think that by using literals you can make the program more space efficient, believing that variables and named constants use slots in memory to store the values associated with them while literals do not.

Actually this is not the case, for two reasons:

- The compiler essentially replaces all occurrences of a named constant with the literal it represents. Thus, named constants are equivalent to literals when the program is running, taking no more space than literals.
- Except for some special, commonly occurring literals such as 0, literals are also stored in memory slots. This is because the machine language does not understand all possible literals. The compiler replaces all occurrences of literals with the addresses of memory slots storing their values. (In fact, some languages such as FORTRAN allowed the value stored in the memory slot to be change by the program, which would, for instance, allow the literal `2.2` to actually mean the value `3.3`!)

In summary, if you are asked to make your code as space efficient as possible, you should not try to use literals instead of named constants.

However, reducing typing is not the main reason for avoiding duplication of code. In fact, another way to prevent retyping of code is to use an editor with copy and paste capabilities. There is another, more, subtle reason for avoiding code repetition, which has to do with the fact that code keeps changing (because of changes in the client's needs and our understanding of the problem). If repeated code must

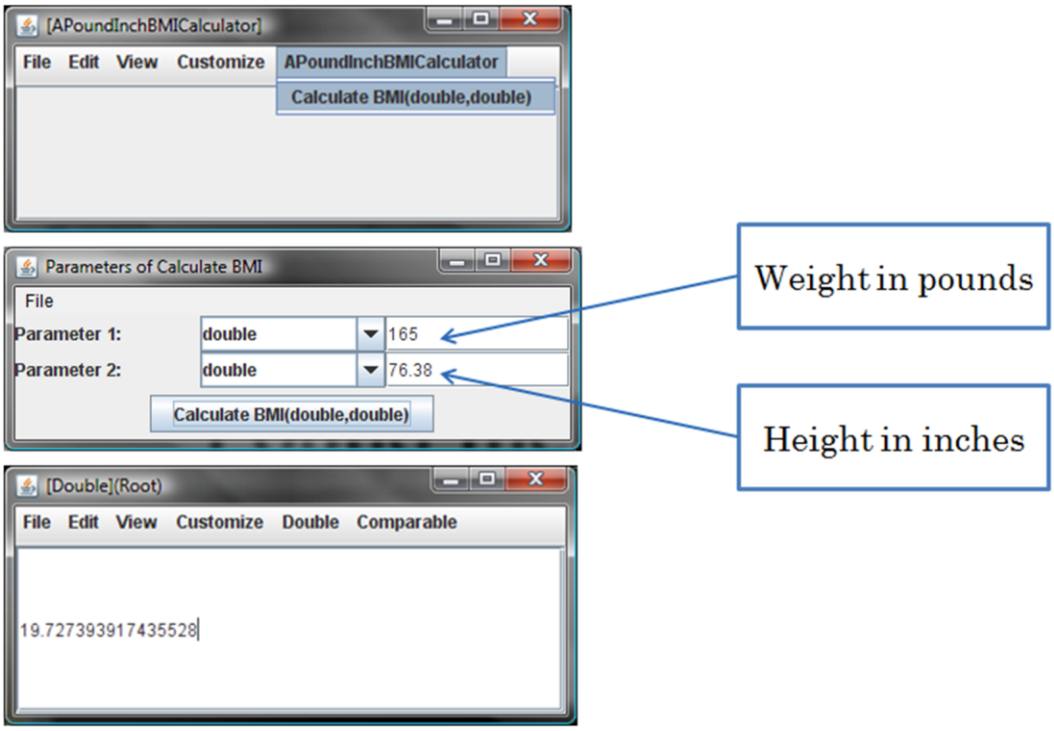

Figure 15. APoundInchBMICalculator

be changed, then we have to locate all occurrences of the code. It is easy to miss a few occurrences, leading to an erroneous program.

To illustrate, after using our example application, our clients may tell us that they would prefer to enter the weight and height in pounds and inches rather than kilograms and meters. If we reused `BMICalculator` in the writing of `AMyBMICalculator`, we have to change only the former. Otherwise, have to change both `BMICalculator` and `MyBMICalculator`, as they both contain the BMI calculation formula.

Multi-Level Algorithms and Stepwise Refinement

Let us consider in more depth the idea of a BMI calculator that works in pound and inches. Instead of changing the existing class `ABMICalculator`, it is better to create a new one, `APoundInchBMICalculator`, that supports these units, so that users have the option of using both sets of units. Figure 15 illustrates the new calculator. Ideally, the `calculateBMI` of `APoundInchBMICalculator` should reuse the code in `calculateBMI` of `ABMICalculator`. The method can take the steps shown to do so:

1. Calculate weight in Kgs from weight in Pounds.
2. Calculate height in Metres from height in inches.
3. Call the method `calculateBMI` of `ABMICalculator` with these values.
4. Return the value returned by this method.

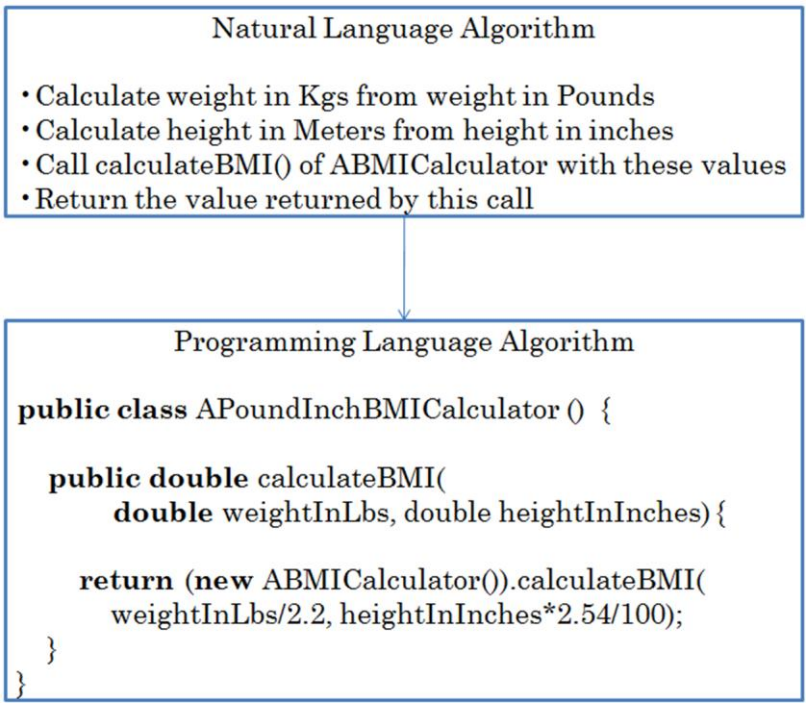

Figure 16. Algorithmic solutions using a natural language (English) and a programming language (Java)

The following Java code implements these steps:

```
public class APoundInchMyBMICalculator (
            double weightInLbs, double heightInInches) {
      public double calculateBMI() {
            return (new ABMICalculator()).calculateBMI(
                  weightInLbs/2.2, heightInInches*2.54/100);
      }
}
```

The English description of the steps we need to take, shown in Figure 16 (top), like the Java code, shown in Figure 16 (bottom), explains the essence of our solution. Such a description is called an *algorithm*. Like Java code, an algorithm is description of the solution to a problem. The description can be encoded using any communication mechanism, graphical or textual. The textual formalism may be a natural or programming language. As we shall see later, often an algorithm is a cross between real code and natural language phrases, and is called *pseudo code*.

An algorithm is a description of not just computer solutions to problems but any well-defined task. For instance, the following is a description of the tasks a professor may perform in a lecture:

1. Enter Class
2. Set up laptop projection

3. Revise topics learnt in the last class
4. Teach today's topics
5. Leave Class

It is often useful to write an algorithm in a natural language or pseudo code before implementing the actual code to be executed. A process such as this that breaks a complex task into multiple simpler steps is called *stepwise refinement*, and is easier to follow and more likely to succeed than one that does not. Writing an algorithm before actual code is one form of stepwise refinement (Figure 16). We will see later other forms.

Algorithms can be described at various ``levels''. For instance, the description given in Figure 16 (top) does not explain how to convert pound and inches to Kgs and metres. The following is a more detailed, 2-level algorithm that does so:

1. Calculate weight in Kgs from weight in Pounds
    a. Divide weight in Pounds by 2.2
2. Calculate height in Metres from height in inches
    a. Calculate height in centimetres from height in inches and divide it by 100 to get height in metres
3. Call the method `calculateBMI` of `ABMICalculator` with these values
4. Return the value returned by this method

However, even this algorithm does not give us all the detail we need – it omits how to compute height in centimeters from height in inches. The following 3-level algorithm does so:

1. Calculate weight in Kgs from weight in Pounds
    a. Divide weight in Pounds by 2.2
2. Calculate height in Metres from height in inches
    a. Calculate height in centimetres from height in inches and divide it by 100 to get height in metres
        i. Multiply height in Inches by 2.54 to get height in centimetres
3. Call the method `calculateBMI` of `ABMICalculator` with these values
4. Return the value returned by this method

Thus, an algorithm need not give all of the details required to implement a program. In fact, even our real-world lecture algorithm leaves out certain important details such as how the topics should be taught, how one distributes the handouts, or what color chalk should be used. Thus, in our stepwise refinement of a solution, we may create multiple levels of algorithms before writing real code.

Multi-Level Code and Internal Calls

Our Java implementation, however, does not reflect multiple levels as it was created directly from the single-level algorithm. In fact, it consists of a single line of Java code. The more units there are in a program, the more (a) understandable it becomes, as people can understand the details added by each

```
public class APoundInchBMICalculator  {

   public double calculateBMI(
         double weightInLbs, double heightInInches) {
      return (new ABMICalculator()).calculateBMI(
         toKgs(weightInLbs), toMetres(heightInInches));
   }
   public double toMetres(double heightInInches) {
      return toCentiMetres(heightInInches)/100;
   }
   public double toCentiMetres(double heightInInches) {
      return heightInInches*2.54;
   }
   public double toKgs(double weightInLbs) {
      return weightInLbs/2.2;
   }
}
```

Figure 17. Implementing each step of the 3rd-level algorithm in a separate method

level incrementally, and (b) reusable it becomes, as there are more parts that can be reused independently. Figure 17 shows an implementation that has a separate method for each level of the 3-level algorithm.

Here, the `calculateBMI` method does not directly convert its parameters to Kgs and metres - instead, it asks the methods `toKgs` and `toMetres` to do so. The method `toKgs` directly converts pounds to Kgs. The method `toMetres`, on the other hand, calls `toCentiMetres`, to convert inches to centimeters, and then converts the centimeters to metres.

Figure 18 illustrates this call sequence when the user calls `calculateBMI` of `APoundInchBMICalculator` with the arguments 165 (lbs) and 70(inches). The method calls `toKgs` to convert 165 lbs to 75Kgs. It then calls `toMetres` to convert 70 inches to 1.778 metres. `toMetres`, in turn, calls `toCentimetres` to convert 70 inches to 177.8 centimetres. Finally, calls `calculateBMI` of `APoundInchBMICalculator` calls `calculateBMI` of `ABMICalculator` with the arguments 75 (Kgs) and 1.77 (metres), and returns the value returned by the latter.

Thus, the process of calling method is very similar to what we saw in the previous examples. The main difference is the complexity of the interaction. In the previous examples, a method called a single other method. Here, three different methods are called by `calculateBMI` of `APoundInchBMICalculator`: `toKgs`, `toMetres`, and `calculateBMI` of `ABMICalculator`. Moreover, previously, a called method directly computed the result. Here, the called method, `toMetres`, calls another method, `toCentimetres`, to compute the result requested by its caller, `calculateBMI` of `APoundInchBMICalculator`.

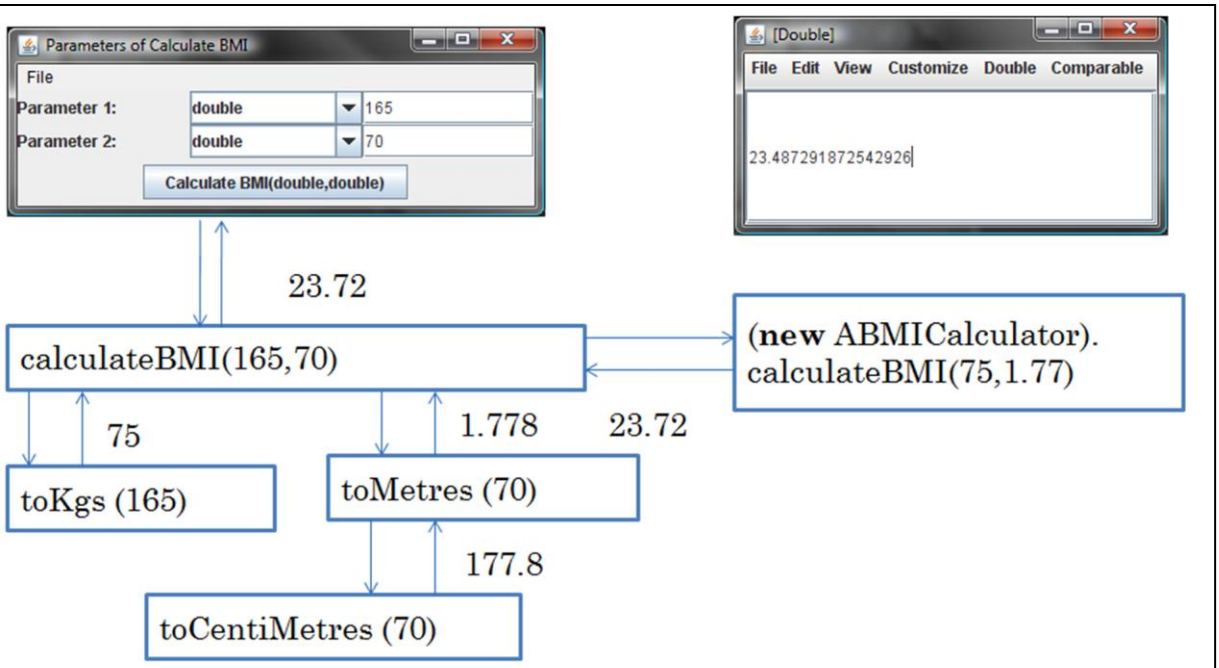

Figure 18. Call graph of calculateBMI in APoundInchBMICalculator

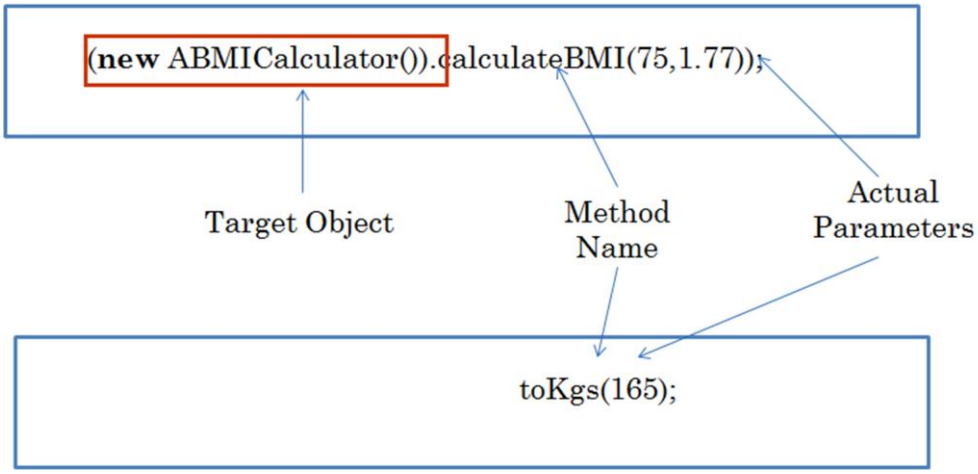

Figure 19. Internal vs. External method call syntax

Notice the difference in syntax between the calls to `calculateBMI` of `ABMICalculator` and `toKgs`, made by the instance of `APoundInchBMICalculator`. The former is an external call, that is, a call made by a method of an object to a method of *another* object. The latter, however, is an internal call, that is, a call made by a method of an object to a method of the same object. In other words, in an external call, the caller and callee methods belong to different objects while in an internal call, they belong to the same object. An external call must always specify the target object. An internal call, on the other hand, need not specify the target object. Figure 19 shows the similarity and difference between internal and external calls.

This version of `APoundInchBMICalculator` is the first example to show that a class can have multiple methods. As we will see later, this is typically the case with most classes. As the previous

```
public class APoundInchBMICalculator  {

   public double calculateBMI(
         double weightInLbs, double heightInInches) {
      return (new ABMICalculator()).calculateBMI(
         toKgs(weightInLbs), toMetres(heightInInches));
   }

   public double toMetres(double heightInInches) {
      final double CMS_IN_METRES = 100;
      return toCentiMetres(heightInInches)/ CMS_IN_METRES;
   }
   public double toCentiMetres(double heightInInches) {
      final double CMS_IN_INCH = 2.54;
      return heightInInches* CMS_IN_INCH;
   }
   public double toKgs(double weightInLbs) {
      final double LBS_IN_KG = 2.2;
      return weightInLbs/LBS_IN_KG;
   }
}
```

Figure 20. Removing all potentially magic numbers

implementation of this class showed, it was not necessary to create multiple methods in the class. Even though this version is longer, it is probably clearer as independent parts are in separate methods appropriately named to convey their function. Moreover, it is more reusable as the methods to convert pounds to Kgs and inches to metres and centimeters can be reused in other classes. For example, another class can reuse the code to convert pounds to Kgs:

```
(new APoundInchBMICalculator()).toKgs(165);
(new APoundInchBMICalculator()).toMetres(70);
(new APoundInchBMICalculator()).toCentiMetres(70);
```

Such reuse was not possible in the first version as these three conversions are all implemented by a single method. As a result, it is not possible to reuse them independent of BMI calculation.

Natural vs. Human-Created Constants

In both implementations, however, we directly used literals, 2.54, 2.2, in the conversions rather than named constants. Are we guilty of using magic numbers? What is magic may depend on the audience. To some readers of code, the numbers 2.2 and 2.54 may not have been magic. To others, who do not know that 100 centimeters make a meter, the number 100 may be magic. The reason why there is ambiguity here is that these are fundamental numbers defined by nature. When a number is human invented, it is more likely to be a magic number. In the statement:

```
return hoursWorked*hourlyWage + 50;
```

the number 50 is probably a magic number that should be replaced by a named constant:

```
return hoursWorked*hourlyWage + BONUS;
```

Even if it was clear to everyone, from this context, that the literal 50 represents the bonus, there is still an important advantage in defining a named constant for it. The bonus might have been used in other parts of the program. Like most human-created numbers, it may change. If we do not define a named constant for the bonus, we must locate and change all occurrences that use it. If we defined a named constant for it, all we have to do is change the place it is declared. As the values of fundamental numbers do not change, there is less reason for defining named constants for such numbers. On the other hand, doing can help people who do not remember the values of these numbers.

In summary, it is best to define named constants for human-invented numbers because they are likely to change and be magic numbers to the readers of the program. Whether you do so for natural numbers depends on the readers of your programs. In case of doubt, defined named constants for them, as shown in Figure 20.