COMP 110/401*

Prasun Dewan[1]

2. Objects

Now that we have a model of how the computer works, we can address the business-at-hand: how do we program the computer. Using two simple, though realistic, examples, this chapter will explain some of the basic elements of a Java program. It will introduce the concept of style in programming and identify some of the basic errors to guard against. It will also outline the process of running a program, that is, converting static program text into active code. After studying this chapter, you will be able to write a program and interact with it.

This chapter is meant for both Comp 110 and Comp 401 students. The names of sections that comp 110 students can ignore have a * next to them.

Java Objects vs. Real-World Objects

Recall that one of the strengths of Java is that it allows a program to be composed of smaller structures much as a script can be broken up into smaller units such as sections, paragraphs, and sentences, or a building can be broken up into rooms, doors, walls, windows, etc. The units of a building are physical objects, while the units of a script are abstract. Like the latter, the units of a program are also abstract. In fact, part of the challenge of programming will be to understand these abstractions. To make programming more intuitive (and powerful), Java and other object-based programming languages provide abstractions, called *objects,* which are modeled after physical objects. Coding in Java consists mainly[2] of defining and interacting with these program objects.

Since program objects are created by human beings, they are more like manufactured physical objects, such as cars and bicycles, rather than natural objects such as trees and rocks. We interact with a (manufactured) physical object by performing different kinds of operations on it. For instance, we accelerate, brake, and steer a car (Figure 1 top). The set of operations we can perform on the car is determined by the factory that manufactured or *defined* it. Defining a new kind of car, thus, involves constructing a new factory for it.

---

[2] In a *pure* object-based programming language such as Smalltalk, programming consists exclusively of defining and interacting with objects. As we shall see later, Java is not such a language, since it provides both object-based and traditional programming. Other books on Java start with traditional programming, whereas, here, we are starting with object-based programming.

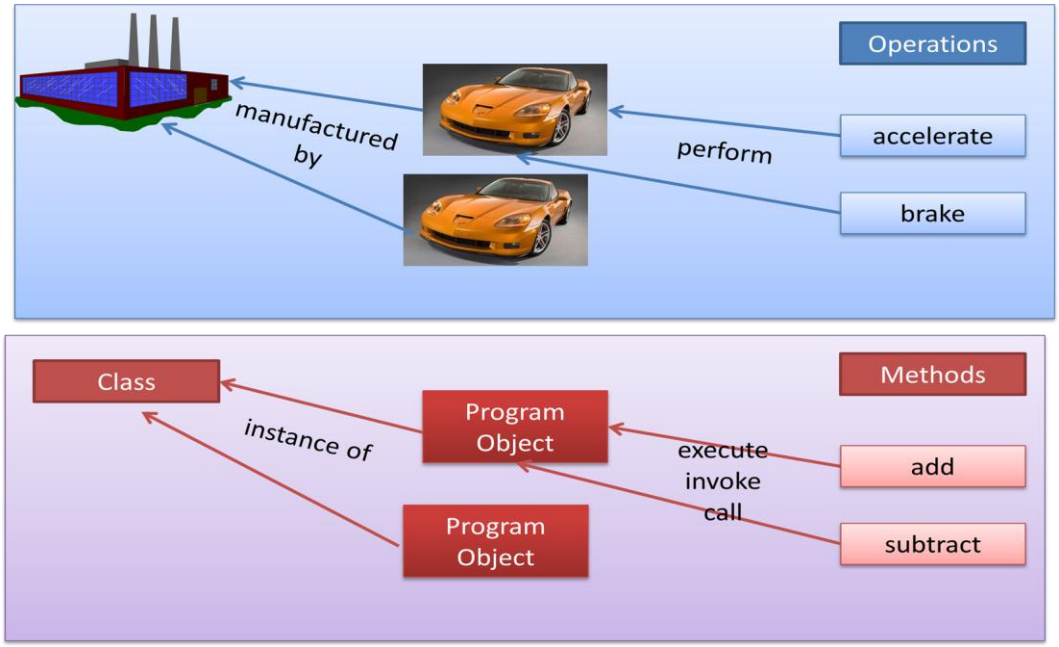

Figure 1. Manufactured physical objects vs. program objects

Table 1. Java vs. real-world

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

Similarly, we interact with a program object by performing different kinds of operations on it. Performing an operation on the program object is also called *invoking* or *executing* or *calling* the operation (Figure 1 bottom). The operation itself is called a *method*. The methods that can be invoked on an object are determined by the *class* of the object, which corresponds to the factory that defines the blueprint of a manufactured physical object. Defining a new kind of computer object, then, involves creating or *defining* or *declaring* a new class for it. An object is called an *instance* of its class. Just as a car can be manufactured on demand by its factory, a computer object can be created on demand by *instantiating* its class. The reason for choosing the term "class" for a computer factory is that it classifies the objects manufactured by it. Two objects of the same class are guaranteed to have the same behavior, much as two cars produced by the same car factory are expected to work in the same way. Table 1 shows the correspondence between these Java concepts and real-world entities.

```
public class ASquareCalculator
{
    public int square(int x)
    {
        return x*x;
    }
}
```

Figure 2. The class `ASquareCalculator`

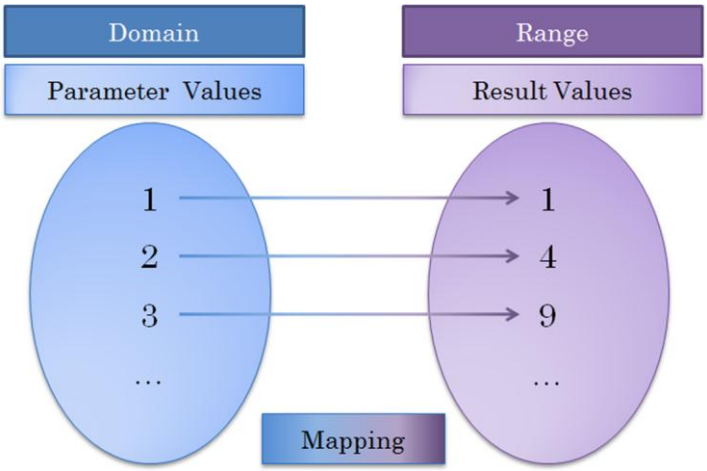

Figure 3. The nature of the `square` **function**

A Simple Class

To make these concepts concrete, let us consider a simple class definition shown in Figure 2. The class is named `ASquareCalculator` and it defines a single method, `square`. This method is a (computer) *function*[3], which is like a mathematics function in that it maps a set of values, called the domain, to another set of values, called the range. The two occurrences of `int` here indicate that both the domain and range of the function is the set of integers. The line

```
return x*x;
```

indicates that an integer x in the domain is mapped to the square of x (that is, x multiplied with itself) in the range. A domain value to be mapped is called a *parameter* of the function and the range value to which it is mapped is called the *result* returned by the function. Figure 3 illustrates the nature of the `square` function. As Figure 3 shows, each of the domain or parameter values is mapped to its square in the range or result values.

The function defined by the class could have also been specified using a more familiar syntax used in mathematics, such as

[3] We will see the other kind of method, a procedure, in the next chapter.

$$\text{square: } I \rightarrow I$$
$$\text{square}(x) = x^2$$

Why not use this familiar syntax to also write programs? It is difficult to follow this syntax literally while writing a program because it is designed for handwritten rather than typed text. For instance, in the above example, it is easy to put the superscript, ^{2}, by hand but not using a word processor. However, there are programming languages, called *functional languages*, which define syntaxes that are inspired by the mathematics syntax. Unfortunately, Java cannot support such syntaxes because it is far more complex than functional languages, having features that conflict with a functional syntax.

What we have done above is define the blueprint of square calculators. To actually manufacture a square calculator, we must instantiate the class `ASquareCalculator`. We can then ask the newly created instance to perform the square operation. We see below two ways to do this. The first, given in at the end of the section, provides some "magic code" that cannot be properly explained without learning many more concepts such as println(), arrays and class methods. It can be ignored if you do not know these concepts and are a purist who likes to understand everything you use. The second, given in the next section, uses a special interactive tool to instantiate objects and invoke methods in them. The idea of using an interactive tool to invoke code is not new – every functional language - such as Lisp and ML - with which I am familiar, and at least one object-oriented language – Smalltalk – provides such a tool. Even though standard Java does not come with such a tool – non-standard versions of such a tool are provided as part of the Dr. Java and BlueJ programming environments. We will use a different tool, called ObjectEditor, built at UNC, which is independent of the programming environment, and more important, is both a code invoker and a user-interface generator. It will allow us to create sophisticated user-interfaces without writing a line of code!

Interactive Class Instantiation and Method Invocation

Let us see how we can use ObjectEditor.[4] to interactively create a class instance and invoke methods in it. The following picture shows the user interface of ObjectEditor. (You may see more than one user interface of ObjectEditor as (a) it continues to evolve and (b) the user-interface is a function of the OS on which the computer runs.

[4] Unlike the editors you may have used so far, such as Windows Notepad or Microsoft Word, ObjectEditor is not an editor of text. Thus, it is not meant for changing the class. Instead, it allows us to create and manipulate objects. The term, "editor", thus, might seem like a bit of a misnomer, at this point. Think of it as an object "driver". Later, when we study properties, you will see that it can also be used to edit the state of an object.

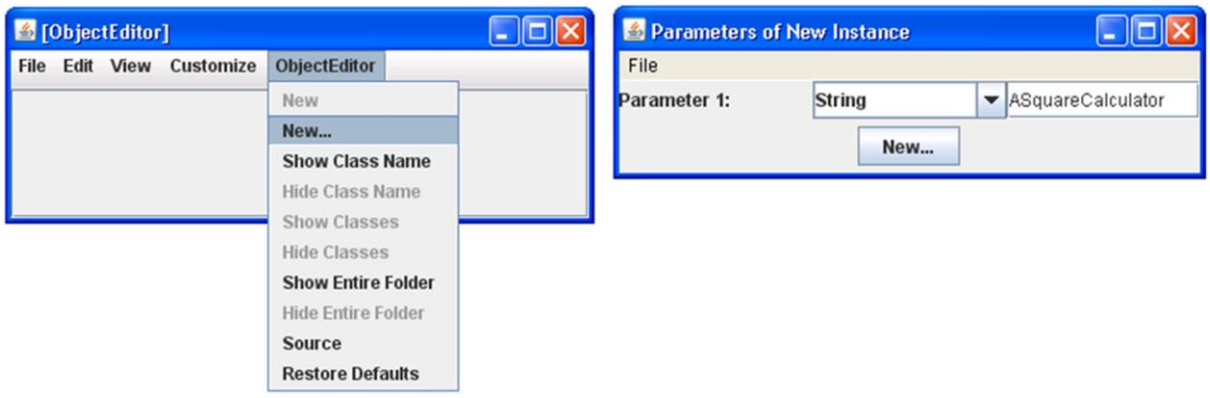

Figure 5. (left) Selecting New… and (right) Entering argument of New command

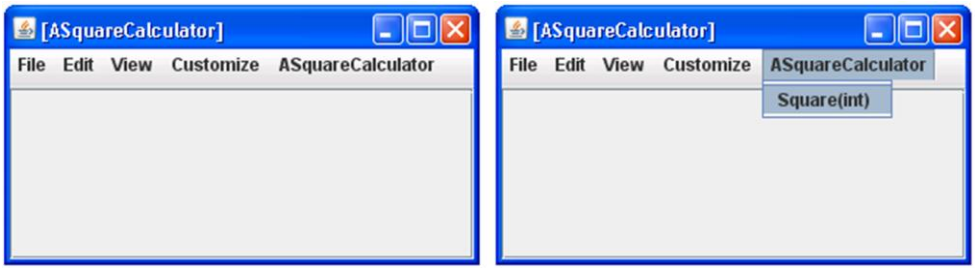

Figure 4. (left) `ASquareCalculator` **instance and (right) calling the** `square` **function of the** `ASquareCalculator` **instance**

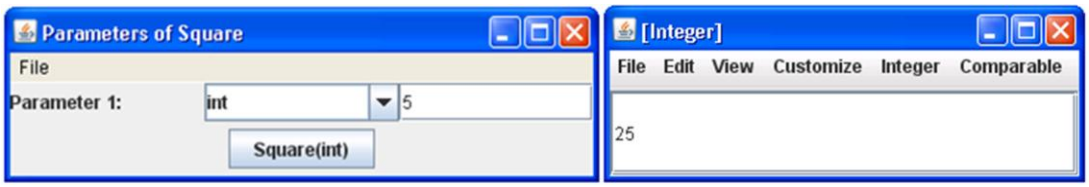

Figure 6. (left) Specifying the parameter and (right) viewing the result

To instantiate our example class, we can execute the New… command, as shown in Figure 5 (left). As the ellipses indicate, the command takes an argument list. Figure 5 (right) shows that argument list consists of a single `String` argument, which is the name of the class to be instantiated.

When it creates a new instance of the class, `ASquareCalculator`, it also creates a new window, shown in Figure 4 (left), to represent the object, which we will refer to as the *edit window of the object*. As you see, it is much like the ObjectEditor window, which represents an instance of the class `ObjectEditor`.

The window provides the menu, ASquareCalculator, to invoke the square operation on the object. It consists of a single menu item, Square(int), which is used to invoke the method of that name provided by the class. The text in braces, (int), after the method name indicates that the method take a parameter (Figure 4 (right)). (Later, we will see methods without parameters and methods with multiple

```
public class ABMICalculator {
    public int calculateBMI (
            int weight, int height)
    {
            return  weight/(height*height);
    }
}
```

```
public class ABMICalculator {
    public double calculateBMI (
            double weight, double height)
    {
            return  weight/(height*height);
    }
}
```

Figure 7. Calculating BMI (left) using integers and (right) using doubles.

parameters). When we select the operation from the menu, a new window, shown in Figure 6 (left), is created, which prompts the user for a parameter value, indicating that an integer is expected this time. If we now supply a value and click on the Square button, ObjectEditor calls the method and displays its result (Figure 6 (right)).

Anatomy of a Class

So far, we have gained a basic understanding of how a class is defined, how a new object of the class is created, and how an operation on the object is invoked. To gain a more precise understanding, let us consider another problem, that of computing our Body Mass Index (BMI), which is our weight in kilogrammes divided by the square of our height in metres. Thus, we need to define a function, let us call it `calculateBMI`, that maps a {weight, height}-pair to a BMI value. In Java, a function cannot be defined in isolation; it must be declared in some class, which essentially groups together a set of related method definitions[5]. Let us call the class of this function `ABMICalculator`. We might be tempted first to write the following class definition as shown in Figure 7 (left). However, this would require the height, weight, and BMI to be converted to integers, which we do not want. What we want, instead, is for each of these to be a real number. Java does understand a real number, which it calls a `double`. Thus we can rewrite the class as Figure 7 (right).

The code follows the pattern we saw in `ASquareCalculator`. This time, however, we will dissect the class definition more carefully so that we can explicitly understand the pattern. Figure 8 shows the various components of the class definition. The line numbers are not part of the class definition, they have been put so that we can easily identify the class components.

The class declaration consists of two parts: a class header and a class body. The class header is the first line of the definition, while the class body is the rest of the definition. The header of a class contains information that is of interest to its users, that is, ObjectEditor (which is just another class) and other classes that instantiate it. The body of a class, on the other hand, gives the class implementation. Think of a class user as a customer of a factory and a class header as information by the factory to potential

[5] In some other languages, such as C++, methods can be defined independently, that is, do not have to part of some class. By requiring methods to be defined in classes, Java encourages cataloging of methods, an important concern from the point of understanding a program.

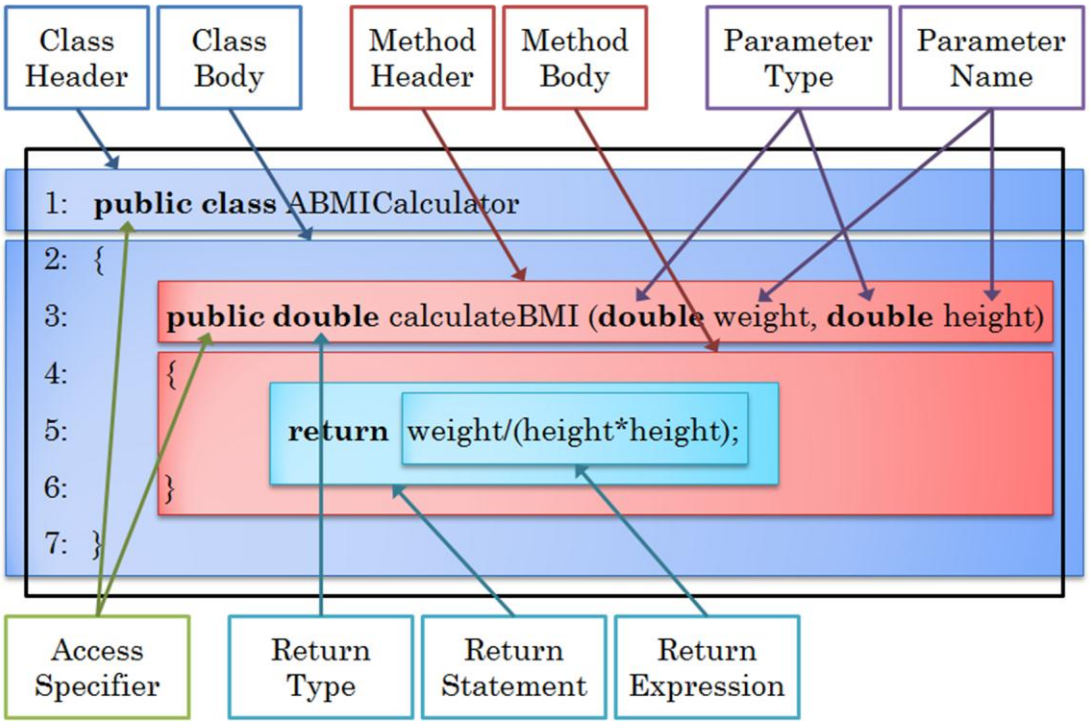

Figure 8. The anatomy of a class

customers on how to order a product manufactured by it. The class body is the process that actually that manufactures the product.

At the minimum, a factory must tell its potential customers that it is a factory, whether they can order products manufactured by it, and the name they should use to refer to it. This is essentially what the three *identifiers* in the class header specify. An identifier is a sequence of letters, numbers, and the underscore character (_) that must begin with a letter. The third identifier here, of course, is the name of the class. The other two are Java *keywords.* A *keyword* is a predefined Java word, which we will identify using **boldface,** that has a special meaning to Java. It is also called a *reserved word*, since it is reserved by the language and cannot be used for identifiers we invent such as the name of a class or method. For instance, the reserved word **double** cannot be used as the name of a class. The keyword **class** says that what follows is a class. The keyword **public** is an *access specifier*. It says that the class can be accessed by any other class - in particular ObjectEditor.[6] If we omitted this keyword, ObjectEditor would not be able to create a new instance of the class.

A method header is essentially information for a customer who has successfully ordered a factory product and is interested in actually using the product, that is, invoking operations on it.

It must tell its potential users that it is a method, whether they can invoke it, what name they should use to refer to it, how many parameters it takes, and what are the sets of values to which the parameters and result belong. This is what the various components of the method header specify. The keyword

[6] If the keyword were omitted, it would be accessible only to classes in its "package". It is too early to talk about how classes are grouped in packages.

public again is an access specifier indicating that the method can be accessed by other classes such as ObjectEditor. If it were omitted, ObjectEditor would not be able to call the method on an instance of the class. It is useful for a class to be public, but not some of its methods, as we shall see later. The next keyword, **double**, is the *type* of the value returned by the function. A type of a value denotes the set to which the value belongs; as mentioned above, the type **double** stands for the set of real numbers. The next identifier, of course, is the name of the method. It is followed by a list of parameter declarations enclosed within parentheses and separated by commas. Each parameter declaration consists of the type of a parameter followed by its name.

A method body defines what happens when the method is invoked. In general, it consists of a series of *statements* enclosed within curly braces and terminated by semicolons. A statement is an instruction to the computer to perform some action. This method consists of a single statement. There are different kinds of statements such as assignment statements, if statements, and while statements. This statement is a *return statement*; we shall study other kinds of statements later. A return statement consists of the keyword return followed by an *expression*, called the *return expression* of the statement. An expression is a piece of Java code that can be evaluated to yield a value. It essentially follows the syntax of expressions you may have used in mathematics, calculators, and spreadsheets; with a few differences, such as the use of the symbol * rather than X for multiplication. We will see other differences later. Examples of expressions include:

```
1.94
weight
weight*1.94
```

A return statement asks Java to return the value computed by its return expression as the return value of the function. Thus, this method body consists of a single statement that returns to its caller the result of evaluating the expression: `weight/(height*height)`.

Program Development

We have seen so far how a class works, but not how it is edited, compiled, or interpreted. How this is done depends on the on the programming environment we use. All programming environments provide a Java compiler and a Java interpreter, normally called javac and java, respectively. Moreover, they provide an editor to compose and change program text, which may be a general-purpose text editor (such as emacs in Unix environment and Notepad in Windows) or a special-purpose editor that understands Java and provides help in entering and understanding Java programs by for, instance, highlighting keywords. In this chapter, we will ignore issues related to specific programming environments – which are addressed in the appendices. In all programming environments, the following steps must be taken (explicitly or implicitly by the programming environment) to create a class that can be instantiated:

1. Create a folder or directory in which the source and object file of the class will be stored. For example, for the class `ASquareCalculator`, create a folder called square and for the class

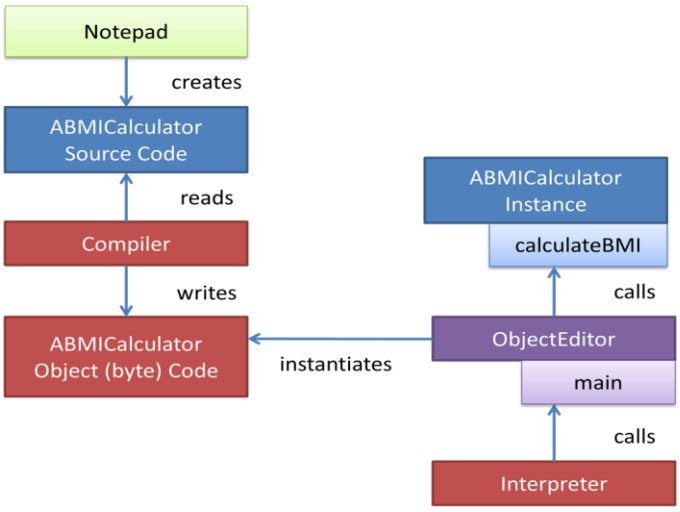

Figure 9. Relationships between Text Editor, Compiler, Interpreter, and ObjectEditor

`ABMICalculator`, create a folder called bmi. The names of the folders do not matter to Java, though, they should indicate the function performed by the class.

2. Create a text file in this folder that contains the source code of the class. Java constrains the name of this file – it should be the name of the class followed by the suffix ".java." Thus, the source code of the class `ASquareCalculator` should be stored in the file ASquareCalculator.java and the source code of the class `ABMICalculator` should be stored in the file ABMICalculator.java.
3. Compile the source code to create object code, also called byte code.

Once the object code of a class is created, it can be instantiated using a main class or ObjectEditor. We will later see in depth how we can develop programs without ObjectEditor.

The following steps must be taken to use ObjectEditor:

4. Ask the Java interpreter to start ObjectEditor, which involves calling a special method in ObjectEditor called the main method.
5. Ask ObjectEditor to create an instance of the class in the manner show in Figure 10 (left).
6. Finally, ask ObjectEditor to execute methods in the class in the manner show in Figure 10 (top-right).

Figure 9 graphically illustrates the main steps in an ObjectEditor-based program development process using the BMI program as an example. It shows the relationships between the four software tools used in this process: the text editor, compiler, interpreter, and ObjectEditor. The text editor creates a class source file that is read by the compiler to create the class object file, which is used by ObjectEditor to instantiate the class. ObjectEditor is started by the interpreter by executing its main method, which is responsible creating the user-interface for instantiating a class and invoking methods in the instance. As Appendix ?? shows, we can run it from a bare bone environment by invoking the command:

```
java bus.uigen.ObjectEditor
```

Other appendices show how it can be started from other programming environment.

Interacting with ABMICalculator

Let us continue with the BMI calculator example. Once the class, `ABMICalculator`, has been compiled, we can use the New … command of ObjectEditor to create a new instance of the class (Figure 10 left and top-right). ObjectEditor displays an edit window to interact with the newly created instance. Instead of the menu ASquareCalculator containing the item Square(int), this time the editor offers the menu, ABMICalculator[7], containing the item Calculate BMI(doule,double) (Figure 10 bottom-right).

[7] If you do not see a menu for the methods of your class, expand the size of the window. If you still do not see the menu, you probably forgot to make any of the methods of the class public, as discussed below.

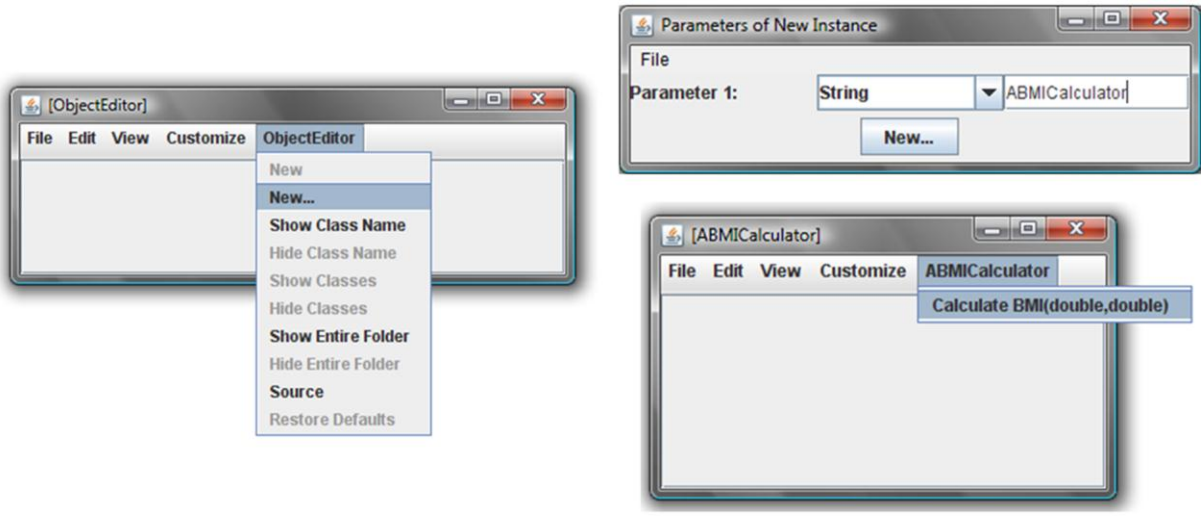

Figure 10. Instantiating ABMICalculator

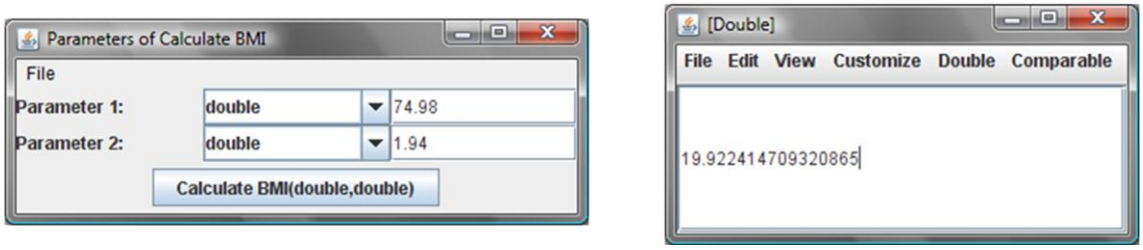

Figure 11. Entering actual parameters and invoking calculateBMI

As before, selecting the menu item creates a dialogue box that prompts for parameter values, displaying the types[8] of the parameters. This time, there are two slots to fill, because the method takes two parameters.

Let us enter 74.98 (Kg) as the first parameter, weight, and 1.94 (meters) as the second parameter, height (Figure 11 left). Clicking on the calculateBMI button executes the function and displays the value returned by it (Figure 11 right).

Formal vs. Actual Parameters, Variables, Positional Correspondence

We have used the word "parameter," above, for two distinct but related concepts. We have used it both for the identifiers, `weight` and `height`, declared in the header of the method, and also the values, 74.98 and 1.94, entered before executing the method. The former, are in fact, called the *formal parameters* of the method definition, while the later are called the *actual parameters* of the method execution.

[8] The fact that the type names appear in pull down-menus may lead you to believe that you have choice regarding what type of parameter value is actually input. This is not the case for int or double parameters, but is the case for some other types of parameters, as we will see when we study interfaces and inheritance.

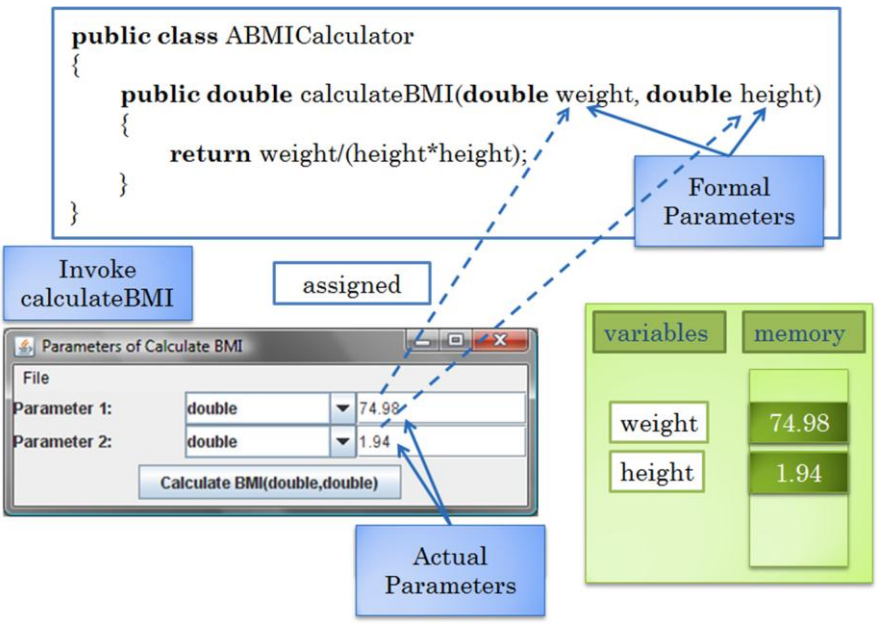

Figure 12. Formal parameters vs. actual parameters

A formal parameter of a method definition is a *variable*, that is, a named slot in memory that can hold a value. It is so called because the value it holds can vary. Storing a value in the memory slot of a variable is called *assigning* the value to the variable.

An actual parameter of a method invocation, on the other hand, is a value that is assigned to the formal parameter when the method is executed. When an expression containing a formal parameter (or any other variable) is evaluated, the variable is replaced with the actual parameter assigned to it.

Thus, when the expression:

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

is evaluated, Java essentially substitutes the two formal parameters with the corresponding actual parameters:

```
74.98 / (1.94 * 1.94)
```

A method can be called several times with different actual parameters; each time different values are assigned to the formal parameters. Thus, if we invoke the method again with the actual parameters, 80 and 1.8, Java performs the substitution:

```
80 / (1.8*1.8)
```

Java uses positions to establish the correspondence between actual and formal parameters, matching the n^{th} actual parameter with the n^{th} formal parameter. Thus, we can change the names (but not positions) of the formal parameters of a method without affecting how we invoke it. An unfortunate side effect is that we must remember the order in which the formal parameters were declared when

invoking the method. Thus, in our example, we must remember that weight is the first formal parameter and height the second one.

As before, we will continue to use the word parameter for both a formal parameter and an actual parameter, relying on context to disambiguate its usage. We will also use the term *argument* for parameter.

Errors

So far, we have assumed that everything goes smoothly in the programming process. As you develop your own code, you will inevitably make errors. As mentioned in the previous chapter, these can be classified into syntax, semantics, and logic errors (Figure 13).

Syntax errors are errors of form. These errors occur when ungrammatical code is input. For instance, if we forget to terminate a return statement with a semicolon:

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

or use the reserved word **double** as a class name:

```
public class double {
```

we have created a syntax error. Syntax errors are reported by the compiler, and in advanced programming environments, by the editor.

Semantic errors occur when the program is grammatically correct but is not meaningful. For instance, if we enter the following return statement:

```
return wght/(hght*hght);
```

we have made a semantic error, since the formal parameters have been named `weight` and `height` and not `wght` and `hght`. We will get a message saying that `wght` has not been declared. An analogous

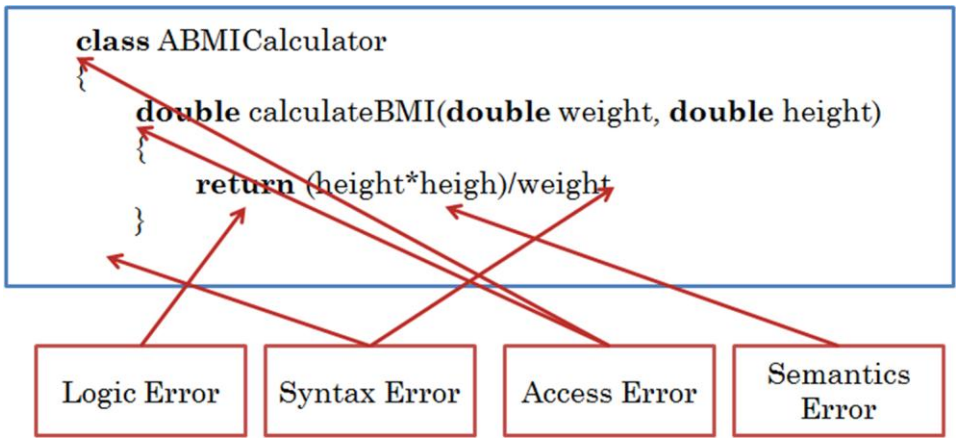

Figure 13. Example errors

situation would occur if Shakespeare created the role `Juliet` but had `Romeo` referring to her as `Julie`!

A program may be syntactically and semantically correct, and thus executable, but still not give the correct results. For instance, if we were to enter:

```
return weight/height;
```

we have made a *logic error*, since we did not square the height. Unlike the other errors, logic errors are not caught by the system - it is our responsibility to check the program for these errors.

A particular form of logic error you must guard against is an *accessibility error*, that is an error in the access specification you have entered for a class or method. For instance, if we omit the **public** access specifier in the declaration of `ABMICalculator`:

```
class ABMICalculator {…}
```

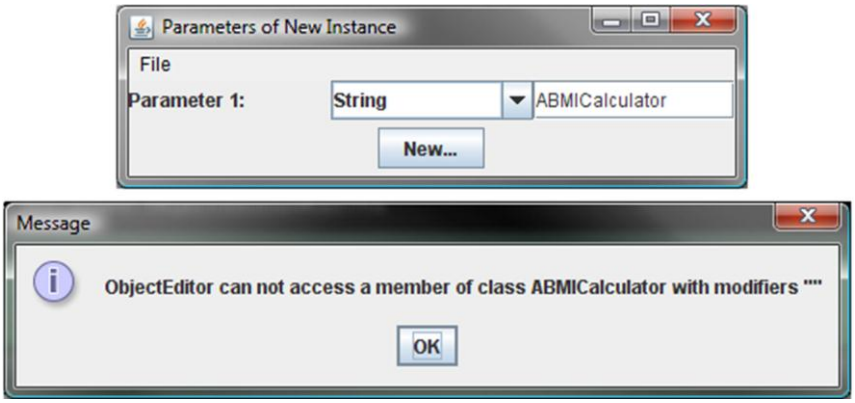

Figure 14. Accessibility error when instantiating a non-public class

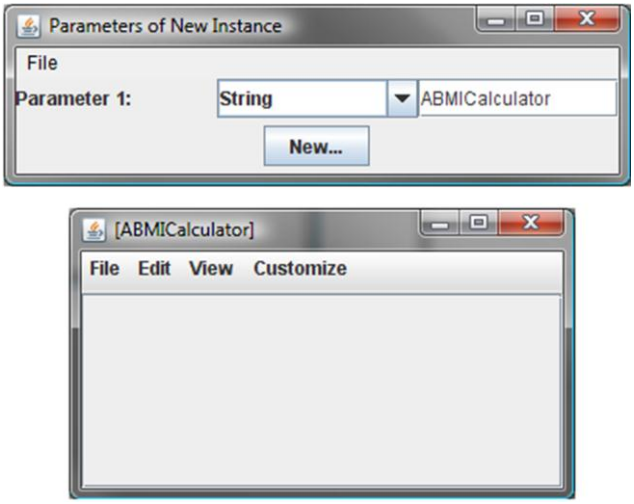

Figure 15. Accessibility error when a class has no public methods

and ask ObjectEditor to instantiate the class, we will get an error message indicating that ObjectEditor cannot access the class (Figure 14). An errors identified while a program is executing is called an *exception*. The particular form or an error made here is called an `IllegalAccessException`.

A similar error occurs if you make the class public, but forget to make its method, `calculateBMI`, public:

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

This time you will not an explicit error message; instead ObjectEditor will not provide a menu item to invoke the method on an instance of the class. Since the only method of `ABMICalculator` is now not public, ObjectEditor does not even create the method menu `ABMICalculator`, as shown in Figure 15 (bottom).

In the two examples above, we made the class/method less accessible than it should have been. Of course, it is possible to make the reverse error of granting a class/method more accessibility than what was intended.

There is another group of errors we need to be concerned with, which are errors users make when interacting with a program. In this case, the program may be perfectly correct but the user essentially does not speak the right interaction language. In the example of `ASquareCalculator`, if we do not use digits for a number, as shown in Figure 16, we have made a *user error*. For now, your program does not need to recover from these errors.

You will need to map the editing, compilation, and runtime error messages generated by the system to the actual errors, which can be a challenging job sometimes and requires some practice. Do not hesitate to ask for help on how to interpret the error messages.

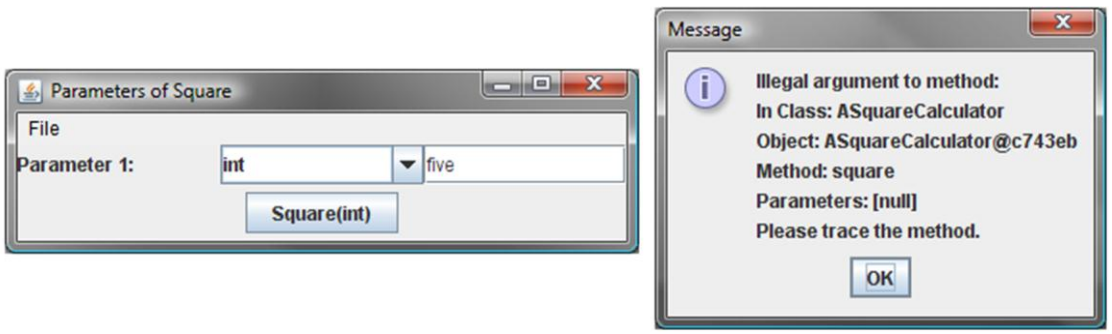

Figure 16. User error

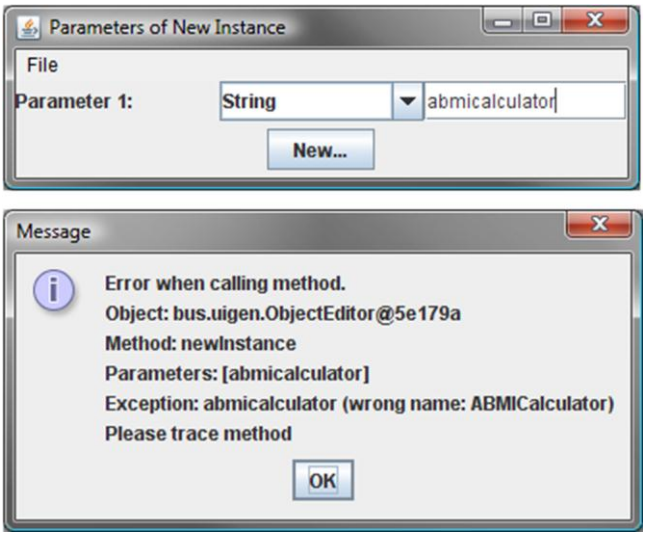

Figure 17. Case matters

Case Matters

Figure 17 shows a likely user error. Here we have entered abmicalculator as the name of the class to be instantiated.

This is an error because whenever we refer to an identifier such as a class name, we must spell it exactly the way it was specified when we declared it. Otherwise Java indicates the exception, `ClassNotFoundException`. Case matters in Java names, that is, two identifiers are considered the same if they are spelled the same way and each pair of corresponding letters have the same case.

This philosophy may seem confusing, especially after using the Windows operating system, which ignores case in the names of files and folders. Java forces us to remember the case we used for each letter in a name. As you gain more experience with programming, you will realize that using case as a basis for creating a unique name can be fairly convenient. It is confusing and error-prone only if we are not careful in following conventions for choosing names.

Case Conventions

By using conventions, we make the program understandable to ourselves, other programmers, and even a tool such as ObjectEditor, as we shall see later. Therefore, let us look at three important conventions that have been established for object-based programs:

- Start the name of a variable (such as a formal parameter) and method with a lower case letter.
- Start the name of a class with an upper case letter.
- To increase readability, we often need to create multi word identifiers. Since spaces cannot be part of an identifier, separate these words by starting each new word (that is, second and subsequent words) with an upper case letter. We followed this rule in the identifiers

`ASquareCalculator`, `ABMICalculator`, and `calculateBMI`. The second rule above, of course, determines the case of the first letter of the first word.

As we study other kinds of namable entities such as constants, interfaces and packages, we will extend these conventions suitably.

Style Principles

These are our first examples of *style principles*. A style principle is a programming rule that makes our code easy to understand. It is not enforced by Java, since it is not necessary for making our programs correct. Therefore, it will be our responsibility to remember and follow all the style principles we identify.

One of the surprising difficulties in programming is choosing appropriate names. Java will allow us to give arbitrary names, since the choice of a name does not influence the correctness of a program. For example, we could have chosen `C` instead of `ABMICalculator` as the name of the second class. Similarly, we could have chosen `x` and `y` instead of `weight` and `height` as the names of formal parameters of `calculateBMI`.

Such names, however, decrease the readability of the program. Therefore, the identifier you use to name a Java entity such as a variable or a class should indicate the specific purpose for which the entity has been declared.

A disadvantage of this principle is that you have to sometimes type long names. The time you spend doing this will greatly reduce the time you, or more importantly someone else, will spend in understanding the program later.

Main and Procedural Class Instantiation and Method Invocation*

We have seen above how we can use ObjectEditor to interactively create instances of classes and invoke methods in them. Let us see here how we can write our own code to do so.

A method must be called by some other method, which in turn must be called by some other method, and so on. This is shown in the figure below. Method R is called by method Q, which is called by method P, and so on.

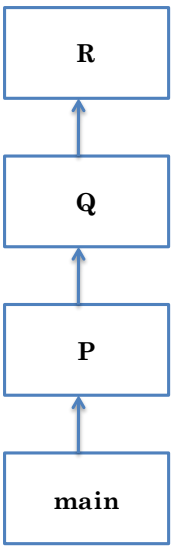

Figure 18 Call Chains

How do we start the sequence of such calls? Java defines a special method, called *main*, to initiate this chain, as shown in the figure. A class defining such a method is called a main class. When we run such a class (using a bare bone or more sophisticated programming environment) this method is called by the Java interpreter. The method has the form:

```
public static void  main (String[] args) {

        ….

}
```

where the ellipsis, …, indicate code we can write to create instances of one or more classes and invoke methods in these instances. Thus, the header or signature of the method is given to us, and we are free to put any code in its body. The header is standardized because the interpreter calls it – if we were to omit the header, the interpreter would have no way to find and call the main method.

This method has components we cannot fully understand at this point – the word **static**[9] and the argument type String[][10]. What is important at this point is that they are necessary in the header of every main method.

The following is an example of a main class and method. It shows how we can write code to instantiate a class and invoke methods in it.

[9] It implies that the method is invoked on the class defining it and not an instance of the class. Thus, in the next figure, it implies that the method is invoked on SquareCalculatorTester rather than an instance of SquareCaoculatorTester.

[10] This type indicates an array or list of Strings. When a main method is called, it is supplied with a list of String arguments entered by the user when the class containing it is run.

```
public class SquareCalculatorTester
{
        public static void main (String[] args)
{
                ASquareCalculator
```

Figure 19 Example of Main Class

The code:

```
ASquareCalculator squareCalculator = new ASquareCalculator();
```

creates a new instance of ASquareCalculator and stores (the memory address of) the instance in the variable, squareCalculator. The type of this variable is the instantiated class. The code fragment:

```
System.out.println (squareCalculator.square(5)))
```

invokes the square() method on the instance referenced by the variable and prints the value returned by the method on the screen. System.out is reference to a predefined object and println() is an operation provided by the object. Thus, two methods are invoked by the code above on two different objects – println() on the object referenced by System.out and the square() on the object referenced by squareCalculator.

Thus, when we run SquareCalculatorTester, its main method will print 25 on the screen. We can run it from the command window by invoking the command:

```
java SquareCalculator
```

Instead of giving the name of the ObjectEditor main class, here we have given the name of our main class.

Similarly, we can write code to create an instance of ABMICalculator and invoke methods in it, as shown below:

```
ABMICalculator bmiCalculator -= new ABMICalculator();
System.out.println(bmiCalculator.calculateBMI(1.9, 95));
```

We will use the term actual parameters to refer to parameter values entered not only interactively, but also programmatically. Thus, in the code above, 1.9 and 95, are actual parameters assigned to the formal parameters, height and weight, before calculateBMI() is executed.

Packages*

As we saw above, a class corresponds to a file. For example, the class ASquareCalculator was stored in the file ASquareCalculator.java.

It is possible to organize a bunch of related classes in a package. Think think of a package as a directory of classes (and as we see later, interfaces) and other packages. In fact, for each package, Java creates a separate operating system directory/folder. The following figure illustrates the use of packages.

```
package math;
public class ASquareCalculator
{
        public int square(int x)
        {
        return x*x;
        }
}
```

```
package main;
import math.ASquareCalculator;
public class SquareCalculatorTester
{
        public static void main (String[] args) {
                ASquareCalculator squareCalculator = new ASquareCalculator()
                System.out.println (squareCalculator.square(5));
        }
}
```

Figure 20 Declaring classes in different packages

Here, classes ASquareCalculator and SquareCalculatorTest are put in packages math and main, respectively, using the first line in the class declaration, which is of the form:

package <package name>

The full name of a class, `C`, defined in some package, `p`, is of the form:

```
p.C
```

just as the name of a file, F, in a Windows directory d is of the form:

```
d\F
```

Thus, in the full name:

```
math.ABMICalculator
```

math is the package name and ABMICalculator is the class name within the package, as the above figure shows. Similarly, in the full name:

```
java.io.BufferedReader
```

`java.io` is the package name and `BufferedReader` is the class name within the package, as the following declaration of the class shows:

```
package java.io;
public class BufferedReader {
        …
}
```

The current convention is to start package names with a lower case letter – in fact the practice seems to be to use only lower case letters, as we see in this example.

Different packages can have classes with the same name, just as different directories can have files with the same name. This is shown in the figure below:

```
package math;
public class ASquareCalculator
{
        public int square(int x)
        {
        return x*x;
        }
}
```

```
package safemath;
public class ASquareCalcula
{
        public long square(int
        {
        return x*x;
        }
}
```

```
package main;
import math.ASquareCalculator;
public class SquareCalculatorTester
{
        public static void main (String[] args) {
                ASquareCalculator squareCalculator = new ASquareCalculator()
                System.out.println (squareCalculator.square(5));
        }
}
```

Figure 21 Class with same short name in different packages

Here, in package safemath, we have defined another class whose short name is ASquareCalculator. This class provides safer math than the previous version of ASquareCalculator by declaring the return type of square() to be long. Like int, long also stores integers, but uses twice as much memory space to do so. In general, the square of an int is not an int – in particular, the square of the largest (and smallest) int is a number larger than it and hence not an int. However, the square of the largest (or smallest) int is a long.

The full name is therefore essential to distinguish these two classes. We can use an import declaration to specify at the beginning of a class or interface definiton the full name of a type we are intested in, we can use the short names subsequently in the remaining code. Thus, as shown in the figure above, by putting the statement:

```
import math.ASquareCalculator
```

at the start of the declaration of the class, BMICalculatorTester, we are able to the short name ASquareCalculator instead of math.ASquareCalculator:

```
ASquareCalculator squareCalculator = new ASquareCalculator();
```

Similarly, if we put the the statement:

```
import java.io.BufferedReader
```

at the start of our class, we can use the short name Bufferedreader instead of java.io.BufferedReader.

It is illegal to import two classes with the same short name because it would then not be clear to which type the short name refers. Thus, the following sequence of statements is illegal:

```
import math.ASquareCalculator
import safemath.ASquareCalculator;
```

A class C, in the same package as some other class, D, does not need to import D to use a short name for it. This is analogous to an HTML file in some folder referring to another HTML file in the same folder with the short name of the latter. This is shown in the following example:

```
package math;
public class ASquareCalculator
{
        public int square(int x)
        {
        return x*x;
        }
}
```

```
package math;
public class SquareCalculatorTester
{
        public static void main (String[] args) {
                ASquareCalculator squareCalculator = new ASquareCalculator()
                System.out.println (squareCalculator.square(5));
        }
}
```

Figure 22 Two classes in the same package can use short names for each other without imports

This time, the main class is the same package as ASquareCalculator. As a result, it does not have an import statement.

Even if two classes are in the same package, they can refer to each other without importing each other. However, in this case, they must use full names. This is shown in the figure below:

```
package math;
public class ASquareCalculator
{
    public int square(int x)
    {
        return x*x;
    }
}
```

```
package main;

public class SquareCalculatorTester
{
        public static void main (String[] args) {
                math.ASquareCalculator squareCalculator = new
math.ASquareCalculator();
                System.out.println (squareCalculator.square(5));
        }
}
```

Figure 23 Classes in different packages use full names instead of import

Here, we have again put the two classes in different packages. This time, instead of importing ASquareCalculator, uses the full name of ASquareCalculator when it needs to refer to ASquareCalculator:

math.ASquareCalculator squareCalculator = **new** math.ASquareCalculator();

If we do not explicitly put a class in a package, then it is put in a nameless *default package* predefined by Java. Classes in the default package have the same short and full names. This is illustrated in the figure below.

```
public class ASquareCalculator
{
        public int square(int x)
        {
        return x*x;
        }
}
```

```
public class SquareCalculatorTester
{
        public static void main (String[] args) {
                ASquareCalculator squareCalculator = new ASquareCalculator()
                System.out.println (squareCalculator.square(5));
        }
}
```

Figure 24 Classes in the default package use short names for each other which are the same as their full names

In this example, both classes are in the default package. As a result they can use short names for each other without imports, which are the same as their full names.

The following figure shows the case in which the referencing class is in the default package and the referenced class is in some other package.

```
package math;
public class ASquareCalculator
{
        public int square(int x)
        {
        return x*x;
        }
}
```

```
import math.ASquareCalculator;
public class SquareCalculatorTester
{
        public static void main (String[] args) {
                ASquareCalculator squareCalculator = new ASquareCalculator()
                System.out.println (squareCalculator.square(5));
        }
}
```

Figure 25 Class in default package needs import/full name to refer to class in some other package

In this case, the referencing class must use either an import of or the full name of the referenced class. However, in the reverse case in which the referenced class is in the default package and the referenced class is in the default package, no import or long names are necessary, as the short name of the referenced class is the same as its long name. This is shown in the figure below:

```
public class ASquareCalculator
{
        public int square(int x)
        {
        return x*x;
        }
}
```

```
package main;
public class SquareCalculatorTester
{
        public static void main (String[] args) {
                ASquareCalculator squareCalculator = new ASquareCalculator()
                System.out.println (squareCalculator.square(5));
        }
}
```

Figure 26 Short name of class in default package same as its long name

Starting ObjectEditor Programmatically

Now that we understand packages, we can start ObjectEditor programmatically, that is, from code we write. ObjectEditor is a class in the package bus.uigen. Thus, its full name is bus.uigen.ObjectEditor. It provides a method called edit(), which takes an arbitrary argument as a parameter. The method creates the user-interface we saw earlier to interact with an object. Thus, the following main class will create the ObjectEfditor user-interface we saw earlier, reproduced in Figure 27 to interact with an instance of ASquareCalculator().

```
package main;
import math.ASquareCalculator;
import bus.uigen.ObjectEditor;
```

```
public class SquareCalculatorTester
{
        public static void main (String[] args) {
                ASquareCalculator squareCalculator = new ASquareCalculator();
                ObjectEditor.edit(squareCalculator );
        }
}
```

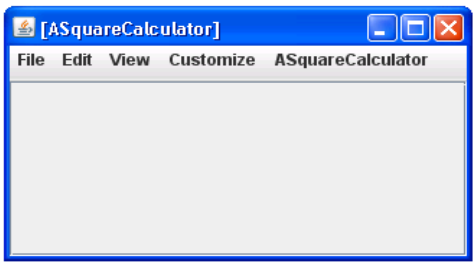

Figure 27 ObectEditor window created from code

When we previously created such a window, we had to use the ObjectEditor interface to interactively instantiate a class. This time we programmatically instantiated the class, and can now use ObjectEditor to interactively invoke methods in an instance of the class.

We can also write our own code to start the ObjectEditor user-interface that allows interactive instantiation of a class. We do so by simply editing an instance of ObjectEditor:

```
        ObjectEditor.edit(new ObjectEditor() );
```

The result will be the creation of user-interface we saw earlier, which is reproduced in Figure 28.

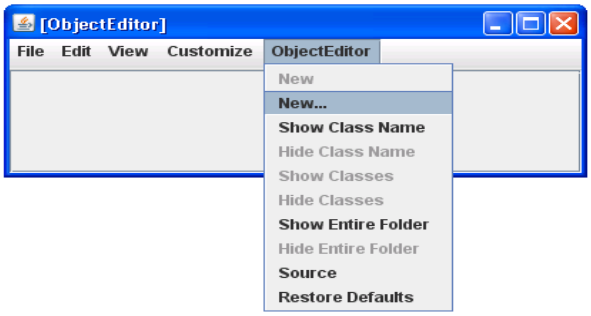

Figure 28 Programmatically creating a user interface to interact with an instance of ObjectEditor

Summary

- Java objects, methods, and classes correspond to real-world objects, the operations that can be performed on them, and the factories that manufacture the objects, respectively.
- A class definition consists of a class header identifying its name and accessibility; and a class body declaring a list of methods.

- A method definition consists of a method header identifying its name, accessibility, formal parameters, and return type; and a method body specifying the statements to be executed when the method is called.
- Formal parameters of a method are variables to which actual parameters are assigned when the method is called.
- The set of values that can be assigned to a formal parameter/returned by a method is determined by the type of the parameter/method.
- The process of writing and executing code we have seen so far consists of declaring one or more methods in a class, compiling the class, asking the interpreter to start ObjectEditor, asking ObjectEditor to instantiate the class, and finally, invoking methods on the new instance.
- By following style principles in writing code, we make the code more understandable to us, others, and tools such as ObjectEditor.

Exercises

1. Define class, method, object, variable, formal parameter, actual parameter, type, statement, return statement, and expression.
2. What are the case conventions for naming classes, methods, and variables?
3. Define syntax error, semantics error, logic error, accessibility error, and user error.
4. Why is it useful to follow style principles?
5. Suppose we need a function that converts an amount of money in dollars to the equivalent amount in cents and can be invoked by ObjectEditor. The following class attempts to provide this function Identify all errors and violation of case conventions in the class.

```
class aMoneyconverter {
       int Cents (dollars) {
            return dollars*100;
      }
}
```

6. Write and test a function, `fahrenheit()`, that takes a parameter an integer centigrade temperature and converts it into the equivalent integer Fahrenheit temperature, as shown in the figures below. Assuming F and C are equivalent Fahrenheit and centigrade temperatures, respectively, the conversion formula is:

```
F = C * 9/5 + 32
```

   Implement the function in the class `ATemperatureConverter`. Use a bare-bones programming environment to develop and execute the class.