

10. Main and Console Input

It is time, finally, to remove the training wheels of `ObjectEditor` and try to write a complete Java program. This means we must ourselves write a main method that implements the user-interface rather than relying on `ObjectEditor`'s main method to do so. We are not quite ready to implement the point and click user-interface provided by `ObjectEditor`, therefore we will implement a different, command-line user interface, which involves use of the console window for entering information and displaying results. We have already seen how to display data in a console window. Now we will see how to receive input from the window. We will implement a user-interface that manipulates instances of the class `ALoanPair` object we created in the last chapter, allowing us to focus on how to program a main method and how to input from the console rather than how to create objects. Nonetheless, the code we create will be complex enough to illustrate the use of developing an algorithm in multiple stages and developing a library that can be called from multiple programs.

Console-based User Interface

It assumes that the user specifies the car loan in terms of its principal, and the house loan in terms of its yearly interest. It prompts the user for these two values, displays the properties of the two loans and their sum, and then terminates.

As we can see, this interface is fairly different from the ones we have seen so far, using the console window for entering data and viewing results. Therefore, we will refer to such a user-interface as a *console-based user-interface* (Figure 1). Such a user-interface is also called a *teletype user-interface*, since it essentially models a typewriter, allowing us to edit only the current line, not the previous lines. It is also termed a *transcript user-interface*, since it shows a transcript or history of our interaction with the application.

`ObjectEditor` provides no help in creating such a user-interface, so we must write our own main method for implementing it. We will put this method in its own class, which will essentially “drive” an instance of `ALoanPair`, invoking methods on the object, much as `ObjectEditor` did in the example of the previous chapter. This class is not quite an editor, since it does not let us edit previous lines, so we will refer to it simply as `ALoanPairDriver`.

¹ © Copyright Prasun Dewan, 2000.

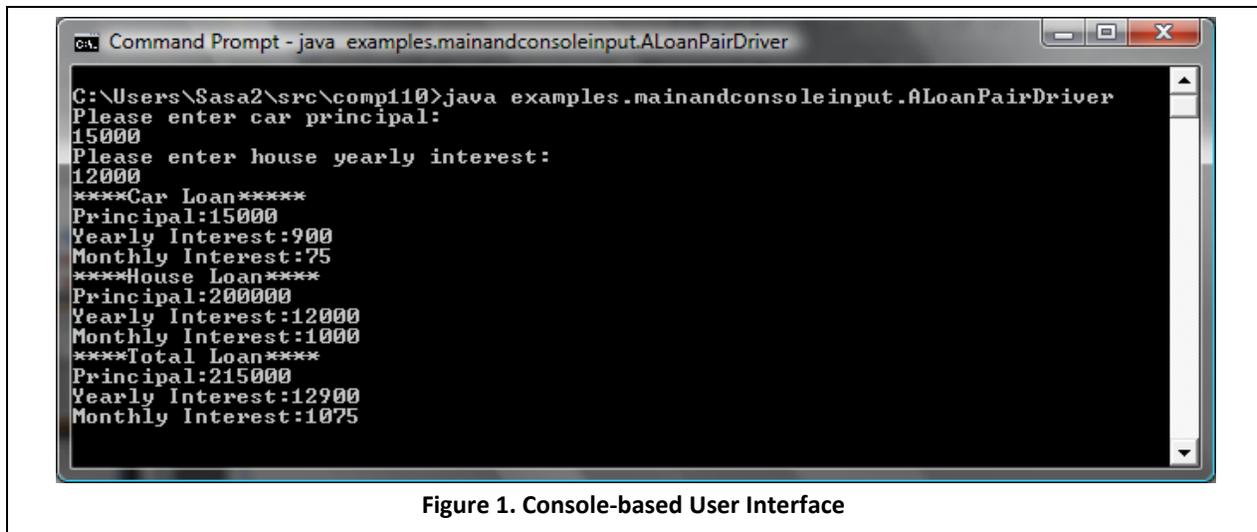


Figure 1. Console-based User Interface

Main Method

At an extremely high-level, the steps to be taken by the main method are:

1. Create `ALoanPair` instance based on the two values entered by the user in the console window.
2. Print the properties of the loan pair object in the console window.

The following main method is an encoding of this algorithm.

```
public static void main (String[] args) {
    LoanPair loanPair = new ALoanPair(readCarLoan(), readHouseLoan());
    print(loanPair);
    bus.uigen.ObjectEditor.edit(loanPair);
    pause();
}
```

Before we exactly understand the statements in it, let us try to see the nature of a `main` method. Recall that this is the method the interpreter calls to start the program. Going back to the theater analogy, the main method is the starting act of a performance. The interpreter invokes it on the class in which it is defined, not on an instance of the class. Thus it must be declared as a class method. Moreover, it must be named `main` and define a formal parameter of type `String[]`. This type is an array, which we will study in depth later. For now, think of it as a list. The interpreter assigns to the parameter a list of string arguments entered by the user when the program is started.

All main methods, thus, have the header shown above. The bodies of the methods, of course, differ based on the algorithm implemented by them. In this example, the body mirrors the algorithm given above. The first statement assumes the existence of functions, `readCarLoan` and `readHouseLoan`, which read input from the console to create `Loan` objects representing a car and house loan, respectively. The statement calls these two functions and uses the two `Loan` objects returned by them to construct a new instance of `ALoanPair`.

The second statement assumes the existence of a procedure for printing the properties of `ALoanPair` instance, and simply calls it to display the output of the program.

The third statement does not have a corresponding step in our algorithm as it has to do with displaying our `loanPair` instance using `ObjectEditor`. The `edit` method of `ObjectEditor` accepts takes an object as a parameter and displays the edit window for the object. The syntax of the `edit` method call may seem somewhat strange. We will discuss this syntax later. For now, just read the statement as an invocation of the `edit` method on `ObjectEditor`. The result is the same as when interactively instantiating a class and displaying the resulting object using `ObjectEditor`, which is the approach we have taken so far. The difference in our current example is that we have instantiated `LoanPair` programmatically, and we want to only display the `LoanPair` object using `ObjectEditor`.

When you display an object by invoking the `edit` method on `ObjectEditor`, any updates you make to your object programmatically will not be automatically displayed in the edit window for the object (later, we will see how to resolve this and have the edit window always reflect the up-to-date property values of an object, whether the properties values are modified programatically or interactively). To display the latest property values in the edit window, you must select `Refresh` option from the `View` menu in `ObjectEditor`.

The fourth statement also does not have a corresponding step in our algorithm and has to do with visual programming environments such as Eclipse. Once a program terminates, these environments destroy its console window². Since a program outputs data fairly fast, a user may not get a chance to view the output before the console window disappears. Therefore, we have put an extra statement in the method, which makes the program pause, that is, block until the user presses a key.

In this method, the statement:

```
LoanPair loanPair = new ALoanPair(readCarLoan(), readHouseLoan());
```

might seem a bit confusing. It is an example of *method chaining*, where the result computed by a method invocation is provided as an argument to another method invocation. Thus, the results of `readCarLoan` and `readHouseLoan` are used as arguments to the constructor `ALoanPair`.

The statement is equivalent to the following three statements:

```
Loan carLoan = readCarLoan();  
Loan houseLoan = readHouseLoan();  
new ALoanPair(carLoan, houseLoan);
```

Method chaining allows us to write fewer statements, but can be confusing to a beginning programmer. It also makes single-step debugging, discussed below, less effective, since multiple actions occur in one

² The current version of Eclipse v3.3 does not have this problem. Instead of using the operating system's console/terminal window for console-based user-interfaces, Eclipse has its own console window integrated into the overall programming environment. As a result, Eclipse does not close the console window when the application stops running.

debugging step, thereby making it difficult to isolate the effect of individual actions. We have seen simple examples of it earlier, with a function call being used as an actual parameter of another method. This example is more complicated because two function calls are used as actual parameters. Whether you use it depends on how comfortable you feel with it.

We have implemented the main method in terms of four methods, `readCarLoan`, `readHouseLoan`, `print`, and `pause`, that we have not yet declared. This is classic top-down programming. Let us go down a level and see how we can implement these methods.

Reading Input

The algorithm for `readCarLoan` is straightforward:

1. Prompt the user for the car loan principal.
2. Return an instance of `ALoan` constructed from the principal.

It is implemented directly by the following code:

```
public static Loan readCarLoan() {
    System.out.println("Please enter car principal:");
    return new ALoan(readInt());
}
```

Like the main method, this is a class method. Since a class method cannot invoke an instance method, all methods called directly or indirectly by the main method must also be declared as class methods. In this method, we have used the `ALoan` implementation of `Loan` since it allows us to construct a `Loan` object from the principal.

The method `readHouseLoan` is very similar, except that it reads the yearly interest and thus returns an instance of `AnotherLoan` (which, recall, constructs a `Loan` object from the yearly interest):

```
public static Loan readHouseLoan() {
    System.out.println("Please enter house yearly interest:");
    return new AnotherLoan(readInt());
}
```

The two methods assume the existence of an as yet undefined method, `readInt`, that reads an `int` value from the console window and returns it as the result. In most languages, such a function would be predefined, but not so in the case of Java. Therefore, we must develop an algorithm for this function as the next level of our program development.

1. Wait for the user to enter a digit sequence on the next line.
2. Return the `int` represented by the digit sequence.
3. In case the user makes an error or terminates input before entering a valid `int`, print an error message and return 0.

The third step requires some explanation. In the interaction shown in the console window, we assumed that the user does indeed enter a digit sequence for each number. What if the user does not do so, and enters, for instance:

one thousand

Our program is not prepared to parse a string containing non digits and considers this entry an error. Similarly, what if the user (by entering a special character, called the *end of file* mark) signals that no more input will be given? Again, this is an error. In general, error recovery is a tricky issue and beyond the scope of this book. Here, we take the simple approach of assuming the value 0 in the case of either of these two errors.

The code of `readInt` is given below:

```
static BufferedReader inputStream = new BufferedReader(
    new InputStreamReader(System.in));
public static int readInt() {
    try {
        return Integer.parseInt(inputStream.readLine());
    } catch (Exception e) {
        System.out.println(e);
        return 0;
    }
}
```

This code uses several features of Java we have not yet seen.

BufferedReader

The function invocation:

`inputStream.readLine()`

returns the input string entered by the user on the next line. As we see in the console window, a user does not need to put quotes around the input string. The method `inputStream.readLine` expects only strings, and considers the beginning and end of the input line as the string delimiters. Each time the method is called, it collects the next line from the user. Here `inputStream` is a class variable declared as follows:

```
BufferedReader inputStream = new BufferedReader(
    new InputStreamReader(System.in));
```

Thus, `inputStream` is assigned an instance of the class `BufferedReader`, which is provided by Java for reading input. Java also provides the object, `System.in`, for reading input, which is counterpart of the object `System.out` provided for printing output. However, this object treats the console input as a sequence of characters and not a sequence of lines; `inputStream` converts the character sequence passed to its constructor to a line sequence.

In this example, we have used a class, `BufferedReader`, not defined by us. Thus, we do not completely understand how our example works. This is not a problem, since, in order to use a class, all we really need to know is how to instantiate it and what its public methods do. Thus, we can treat `BufferedReader` as simply a “black box” that reads lines for us.

While `BufferedReader` knows how to read strings from the console, it does not know how to convert these strings to `int` values. This is the task of another class, `Integer`³, which provides a class method, `parseInt`, for doing this conversion. Therefore, `readInt` invokes this method, passing it the string returned by `inputStream.readLine` and returns the `int` computed by it.

Importing from a Package

`BufferedReader` is a short name for the full name, `java.io.BufferedReader`. We can use the short name in a class only if, before the class declaration begins, we *import* the long name, that is explicitly specify its name in a declaration beginning with the keyword **import**:

```
import java.io.BufferedReader;
public class ALoanPairDriver { ... }
```

Thus, in any class that must read lines from the console, be sure to insert this import declaration and the `dataIn` declaration we saw above.

The reason why there is a difference between the short name and long name of `BufferedReader` is that it, unlike the types (classes and interfaces) we have created, this class has been put in a *package*. Think of a package as a directory of types and other packages. In fact, for each package, Java creates a separate operating system directory. The full name of a type, `T`, defined in some package, `p`, is of the form:

`p.T`

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

`d\F`

Thus, in the full name:

`java.io.BufferedReader`

`java.io` is the package name and `BufferedReader` is the class name within the package. 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.

³ `Integer` is related to but not the same as `int`. Both are types represent Mathematical integers. However, `Integer` is a class while `int` is a primitive type. We will later study in more depth the relationship between the two.

If we do not explicitly put a class in a package, then it is put in a nameless *default package* predefined by Java. All classes created by us were put in this package. Classes in the default package have the same short and full names.

Different packages can have classes with the same name, just as different directories can have files with the same name. The full name is therefore essential to distinguish these classes. When we use an import declaration to specify at the beginning of a class or interface definition the full name of a type we are interested in, we can use the short names subsequently in the remaining code. It is illegal to import two types with the same short name because it would then not be clear to which type the short name refers.

Exceptions

As mentioned above, two kinds of errors can occur when reading an `int`: a user may prematurely end input or enter a non-digit sequence. The method `InputStream.readLine` detects the first error and the method `Integer.parseInt` detects the second one. Before we see how these particular methods process these errors, let us try to answer the general question: What should a method do when it detects an error? There are two choices.

1. It can handle the error in some application-specific way. This is the approach taken in the design of `readInt`. In case of errors, it returns the value 0.
2. It can “pass the buck” to the method that called it, letting it handle the error. This is the approach taken by `InputStream.readLine` and `Integer.parseInt`.

Java provides an elaborate mechanism for “passing the buck,” which we will not study in depth here. We will study just enough of it necessary to use some standard methods provided by Java.

In Java, an error is represented by an object called an *exception*. The class of the object indicates the kind of error represented by it. For example, an input or output error such as premature end of file is represented by an instance of `IOException`; using a wrong representation for a number is represented by an instance of `NumberFormatException`; and an attempt to invoke a method on a null pointer is represented by an instance of `NullPointerException`, which we saw earlier encountered.

To pass the buck, a method *throws* an exception of the appropriate class, which can be *caught* by the method that calls it. For now, let us ignore how exceptions are thrown; instead, let us look at a simple approach to catch exceptions. The following code fragment shows how this is done:

```
try {
    return Integer.parseInt(inputStream.readLine());
} catch (Exception e) {
    System.out.println(e);
    return 0;
}
```

This is an example of a *try-catch* statement, consisting of a *try block* and a *catch block*. It indicates, through the `try` keyword, that the method is only attempting to execute the statements in the `try`

block – there is no guarantee that it will execute all of these statements because of exceptions. If an exception is thrown by a method invoked in the try block, the exception is “caught” by terminating the try block and transferring control to the catch block. The parameter, *e*, in the header of the catch block contains the value of the thrown exception, which can then be examined by the body. In this example, the body simply prints the exception and returns the value 0.

If we do not enclose the statement

```
return Integer.parseInt(inputStream.readLine());
```

within a try-catch statement, the Java compiler will complain. It will ask us to either catch the exception by putting a try catch block around it or declare a *throws clause* in the method header. For now, ignore the throws clause; always react to the message by putting a try-catch block around the statement about which the compiler issues this complaint.

The following definition of `pause` illustrates another use of try-catch block:

```
public static void pause() {
    try {
        System.in.read();
    } catch (Exception e) {
        System.out.println(e);
    }
}
```

The purpose of this method is to stop the main method from terminating until the user enters some character. The method `System.in.read` is much like `InputStream.readLine` in that it waits for user input – the difference is that it returns the next character rather than the next line. Thus, by calling it, we achieve our goal of blocking the program until the user enters a character. As in `readInt`, there is the chance that the user may enter an end of file character, causing an `IOException`. Therefore, we have enclosed the method call within a try-catch block. Notice that unlike the previous catch block, the catch block above does not return any value. This is because `pause` is a procedure rather than a function. It does not really care about the value of the character returned by `System.in.read`, therefore it does not store its value in any variable or return it.

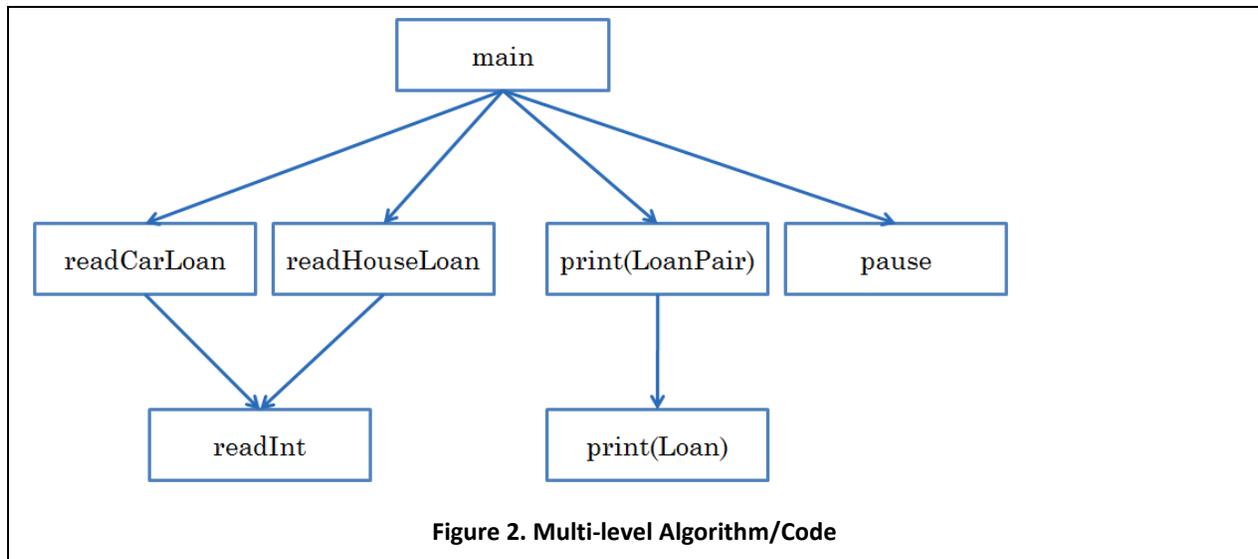
Writing a complete try-catch statement whenever we want to receive input from the user is, of course, tedious. We will soon relieve this tedium by writing a reusable library that does this for us.

Programmer-defined Overloading

To complete our description of `main`, we need to look at the `print` procedure called by:

```
print(loanPair);
```

We can implement this method by assuming that a `print` exists to display a `Loan`, and call it three times for printing the three `Loan` properties of its `LoanPair` argument:



```

public static void print(LoanPair loanPair) {
    System.out.println("****Car Loan****");
    print(loanPair.getCarLoan());
    System.out.println("****House Loan****");
    print(loanPair.getHouseLoan());
    System.out.println("****Total Loan****");
    print (loanPair.getTotalLoan());
}

```

The implementation of the method for printing a `Loan` is similar, printing all the properties of its argument, except that this time, `System.out.println` can be used directly to print the primitive properties:

```

public static void print(Loan loan) {
    System.out.println("Principal:" + loan.getPrincipal());
    System.out.println(
        "Yearly Interest:" + loan.getYearlyInterest());
    System.out.println(
        "Monthly Interest:" + loan.getMonthlyInterest());
}

```

Thus, we have created two new implementation of `print` in this class, one for printing a `Loan`, and other for printing a `LoanPair`. As mentioned before, such overloading of a method name is allowed in Java. When we use the name of an overloaded method in a call, Java determines which implementation to invoke based on the types of the actual parameters of the invocation. Thus, when it sees:

```
print(loanPair)
```

it calls the `print` whose formal parameter is of the type of the actual parameter of this call, `LoanPair`; and when it sees:

```
print(loanPair.getCarLoan());
```

it calls the print whose formal parameter is type `Loan`. Java will not let us define two implementations of a method that take the same types of arguments.

Multi-level Algorithm

We have seen here an example of an algorithm, and the code that implements it, developed in multiple stages. Figure 2 shows the levels of the algorithm and the code.

In this figure, we see only the methods defined in `ALoanPairDriver`, not those we defined earlier in `ALoanPair` and `ALoan`. If we were to include calls to the latter also, we would have even more levels. In general, the more levels in an algorithm, the easier it is understand and get right, since each level can be developed independently of other levels. In this chapter, we have illustrated the kind of top-down thinking required to do a multi-level decomposition.

The complete code of `LoanPairDriver` is given below, showing how the various levels work together.

```

import java.io.BufferedReader;
public class ALoanPairDriver {
    static BufferedReader inputStream = new BufferedReader(
        new InputStreamReader(System.in));
    public static void main (String args[]) {
        LoanPair loanPair = new ALoanPair(
            readCarLoan(), readHouseLoan());
        print (loanPair);
        pause();
    }
    public static Loan readCarLoan() {
        System.out.println("Please enter car principal:");
        return new ALoan(readInt());
    }
    public static Loan readHouseLoan() {
        System.out.println("Please enter house yearly interest:");
        return new AnotherLoan(readInt());
    }
    public static int readInt() {
        try {
            return Integer.parseInt(inputStream.readLine());
        } catch (Exception e) {
            return 0;
        }
    }
    public static void print (LoanPair loanPair) {
        System.out.println("****Car Loan****");
        print(loanPair.getCarLoan());
        System.out.println("****House Loan****");
        print(loanPair.getHouseLoan());
        System.out.println("****Total Loan****");
        print (loanPair.getTotalLoan());
    }
    public static void print(Loan loan) {
        System.out.println(
            "Principal:" + loan.getPrincipal());
        System.out.println(
            "Yearly Interest:" + loan.getYearlyInterest());
        System.out.println(
            "Monthly Interest:" + loan.getMonthlyInterest());
    }
    public static void pause() {
        try {
            System.in.read();
        } catch (Exception e) {
            System.out.println(e);
        }
    }
}

```

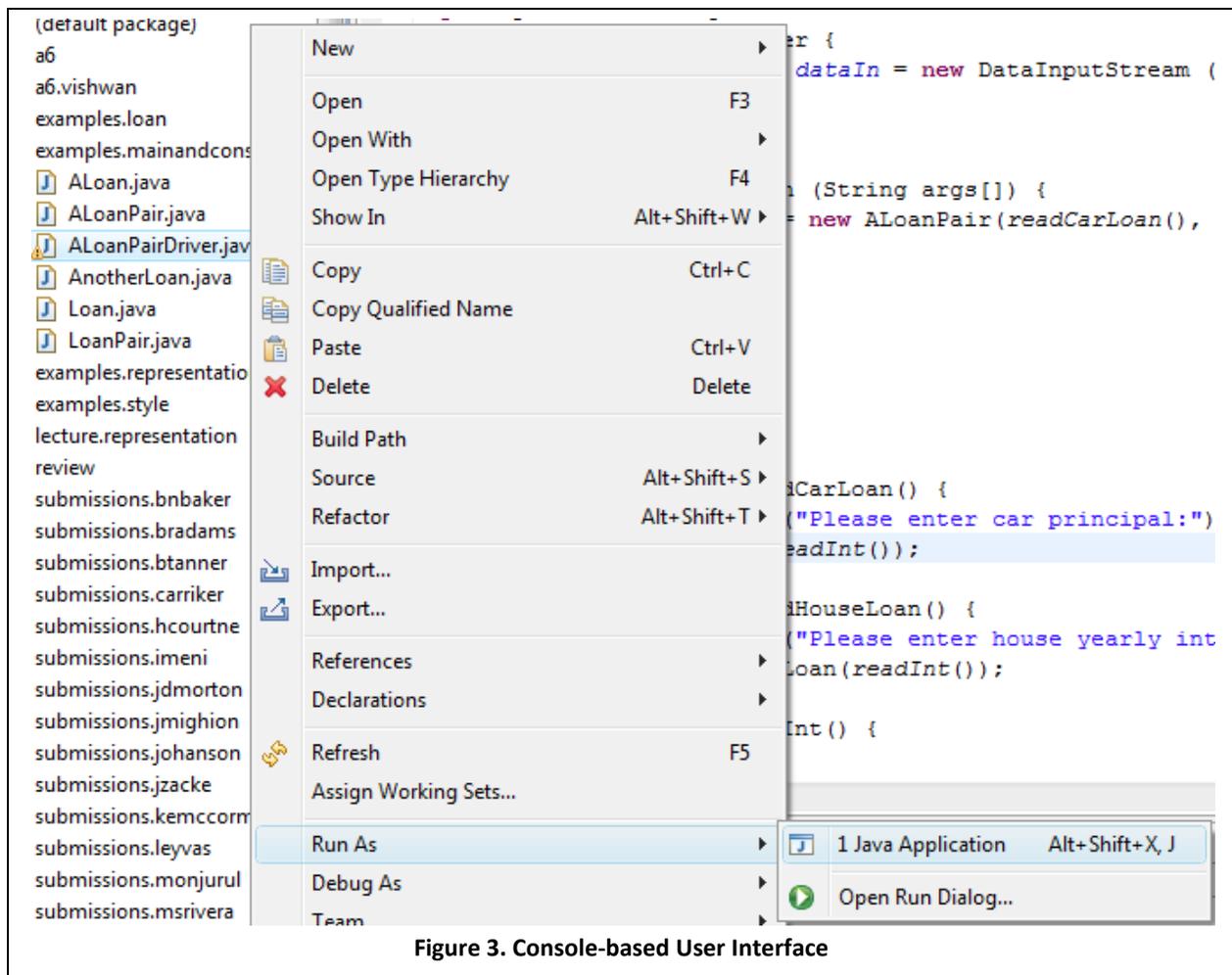


Figure 3. Console-based User Interface

Running Main

Now that we understand how a main method is written, let us see how it is invoked. Recall that the class in which it is defined takes the place of `ObjectEditor`. Thus, if we are using the command interpreter, instead of typing the command:

```
java ObjectEditor
```

we should now type the command:

```
java <Main Class>
```

where `<Main Class>` contains the main method. In this example, we should enter:

```
java ALoanPairDriver
```

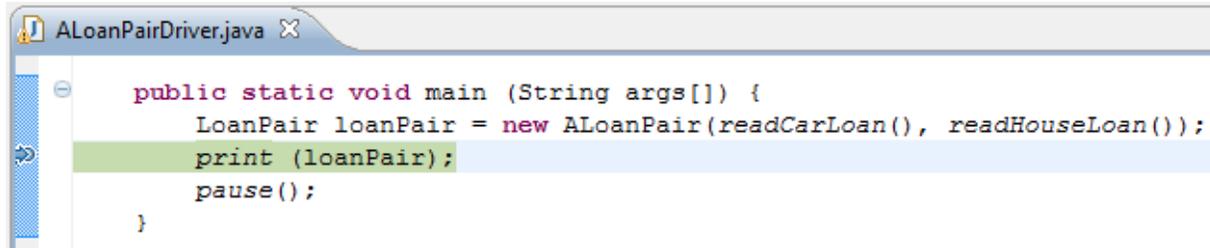
The class following the `java` command is the one in which the interpreter looks for the main method.

If we are using Eclipse, all we have to do is execute the `Debug` command in the `Run` menu. Alternatively, you can right-click on `ALoanPairDriver` in the `Package Explorer` window and select `Java`

```
import java.io.DataInputStream;
public class ALoanPairDriver {
    static DataInputStream dataIn = new DataInputStream (System.in);

    public static void main (String args[]) {
        LoanPair loanPair = new ALoanPair(readCarLoan(), readHouseLoan());
        Line breakpoint:ALoanPairDriver [line:9] - main(String[])
        pause();
    }
}
```

Figure 4. Placing a breakpoint in Eclipse



```
ALoanPairDriver.java X
public static void main (String args[]) {
    LoanPair loanPair = new ALoanPair(readCarLoan(), readHouseLoan());
    print (loanPair);
    pause();
}
```

Figure 5. Eclipse debugger view when the debugger hits a break point

Application from the `Debug As` menu (Figure 3). We can also execute the `Run` command, but it will not give us Eclipse debugging support. We advocated the use of it when running `ObjectEditor` to speed up performance. Our main methods, being far less complex than the main method of `ObjectEditor`, should not create any performance problems, and thus the `Debug` command is the better option for them.

Tracing Using a Debugger

We saw earlier the use of print statements to trace the execution of a program. Debuggers provide an alternative mechanism to do the same, which does not require the overhead of adding/deleting print statements in order to start/stop tracing the program. Let us look at the Eclipse debugger to illustrate this mechanism.

In order to trace a program, place a break point on the second statement of the `main` method. To do this, double-click, at the height of the second statement of `main`, on the column that is furthest to the left from the code in the code window (Figure 4). When running in debug mode, the debugger stops program execution when it encounters (hits) a break point on a statement (Figure 5). In Figure 5, we see the result of hitting the first break point. At this point, the debugger is ready to invoke the `print` method. Execute the `Step Over` command from the `Run` menu. In this execution mode, the debugger executes one statement at a time, instead of executing the whole program at once. Each subsequent invocation of the `Step Over` command executes one more statement. An arrow displayed on the sidebar indicates the next statement to be executed.

While a program is stopped at some statement, we can inspect the variables visible to that statement by selecting `Variables` from the `Show View` menu item in the `Window` menu. Figure 6 shows what

Name	Value
args	String[0] (id=16)
loanPair	ALoanPair (id=18)
carLoan	ALoan (id=22)
principal	15000
houseLoan	AnotherLoan (id=25)
yearlyInterest	12000

Figure 6. Inspecting variables during pause in execution of program in Eclipse

```

public static void main (String args[]) {
    LoanPair loanPair = new ALoanPair(readCarLoan(), readHouseLoan());
    print (loanPair);
    pause();
}

```

Figure 7. Executing the Step Over debug command when stopped at the print statement in main

```

public static void print (LoanPair loanPair) {
    System.out.println("****Car Loan****");
    print(loanPair.getCarLoan());
    System.out.println("****House Loan****");
    print(loanPair.getHouseLoan());
    System.out.println("****Total Loan****");
    print (loanPair.getTotalLoan());
}

```

Figure 8. Executing the Step Into debug command when stopped at the print statement in main

happens when we make this selection. A special window, called `variables`, is created to display the formal parameter, `args`, and the local variable, `loanPair`, which are the two variables visible to statements in the main method.

As we can see, the window shows the entire physical structure of each of the displayed variables. It shows that `args` is assigned an empty array of `String` values and `loanPair` is assigned an instance of `ALoanPair`, which consists of two instance variables, `carLoan` and `houseLoan`. The variable `carLoan` is assigned an instance of `ALoan` and the variable `houseLoan` is assigned an instance of `AnotherLoan`. The instance of `ALoan` consists of the the instance variable `principal`, whose value is 15000. The instance of `AnotherLoan` consists of a single instance variable, `yearlyInterest`, whose value is 12000. As we step through the statements of the program, this window is updated automatically to reflect changes made to the displayed variables by the executed statements. The display does not show the values of the properties of an object, for which you must rely on `ObjectEditor`.

When execution is stopped at some method call, such as the print statement in Figure 5, what is the “next” statement the debugger should stop at. There are two possible answers : (1) the next statement in the calling method, `main`, (Figure 7), or (2) the first statement in the called method, `print` (Figure 8). The `Step Over` command chooses the first alternative, allowing us to remain at the same level in our algorithm. If we wish the second alternative so that we can go down one level in the algorithm, we must execute another command, the `Step Into` command. The converse of this command, `Step Out`, lets us return to the previous level.

Function vs. Procedure Invocation Again

Now that we have a basic understanding of how `ALoanPairDriver` is written and executed, let us try to understand some subtle points illustrated by it. Compare the ways in which we invoked the function `readInt`

```
return new ALoan(readInt());
```

and the procedure `print`:

```
print(loanPair);
```

The function invocation is used an expression, while the procedure invocation is used as a statement. As we saw before, a procedure invocation cannot be used as an expression, since it does not produce a value. Thus, the following is illegal:

```
System.out.println(print (loanPair));
```

What about a function invocation? Should it be allowed to be used as a statement:

```
readInt();
```

As it turns out, a function invocation is indeed a legal statement in Java, but a program that uses this feature is probably erroneous, since it is ignoring the return value. In fact, beginning programmers have a tendency to make the mistake of ignoring a function return value they need, so it is unfortunate Java does automatically catch this error.

When is it possible to write "working" programs that ignore the return value of a function? Sometimes, the return value is an error code, that is, a special value signifying an error. In languages that do not support exceptions, error codes are the only way for a called method to indicate to its caller that an error occurred. A lazy programmer, hoping there are no errors, may ignore such codes, creating programs that "work" in the absence of errors. This kind of programming is not recommended, and, therefore, many programming languages such as Pascal disallow it.

An example of a defensible use of this feature is the following function invocation in `pause`:

```
System.in.read();
```

Recall that the purpose of this method invocation is to block a program until we provide some input, giving us a chance to view the output. Since the program does not process the character we enter to terminate the program, it can safely ignore it instead of doing a spurious assignment:

```
char c = (char) System.in.read();
```

However, uses such as these are unusual. If your program is not working, be sure to check that you are using all function invocations as expressions.

Returning Vs Printing a Result

Beginner programmers, when asked to write a function that produces a certain result, tend to print the result instead of, or in addition, to returning it to the calling method:

```
public static int readInt() {
    int retVal;
    try {
        retVal = Integer.parseInt(dataIn.readLine());
    } catch (Exception e) {
        System.out.println(e);
        retVal = 0;
    }
    System.out.println(retVal);
    return retVal;
}
```

The feeling is that the result will be “lost” if it is not printed. It is important to realize that every function has a caller, who will be responsible for processing the return value. Thus, we should not print the return value of a function unless there are special reasons for doing so, such as debugging it.

Side Effects

Printing in a function is a *side effect*. We normally think of the effect of a function as computing a return value. A “side effect” is any other effect of the function that can be observed by the user or another method after the function has terminated. Examples of side effects are:

1. Writing output to the console window.
2. Advancing the read cursor in the console window (which is shared by all methods) by reading input.
3. Changing the value of a global variable, which is also shared with other methods in the class.

Side effects can be confusing, and therefore should be avoided as much as possible. Some side effects are more confusing than others: the three side effects above have been listed in the order of their potential for causing harm. Writing output is perhaps the most benign side effect, and is something we often need to do in a function to debug it or report errors. In Java, reading input is also acceptable, since

a method that returns a result based on processing the input stream of characters must be a function⁴. This is the reason that the predefined Java input methods we used in this program, `dataIn.readLine` and `System.in.readChar`, and the three we defined, `readInt`, `readCarLoan`, and `readHouseLoan`, are all functions.

The most dangerous is the last one, and as it turns out, is easily avoided in most cases without significantly reduce programming flexibility. Therefore, you should not change global variables in functions. We will see one exception to this rule later, when we look at an enumeration interface. An exception is allowed in the case of an enumeration interface because, as we will see later, it is very much like the read cursor case.

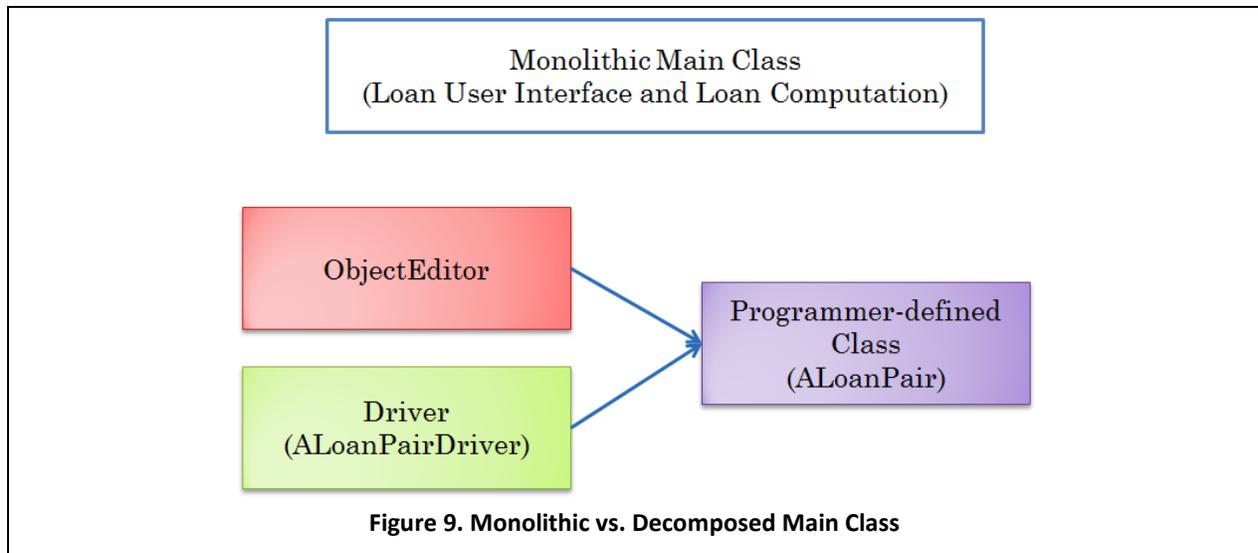
Some programmers feel that even procedures should never change the values of global variables, since it makes them less self-contained. However, this rule unduly reduces the flexibility of what we can program, since, as we have seen, global (instance or class) variables are necessary to code several modern applications such as the spreadsheets.

ObjectEditor-Influenced Decomposition

We have seen in `ALoanPairDriver` our first example of a full program, that is, a program with a main method written by us. Though we did need the need any help from `ObjectEditor` to execute our code, its nature was in fact influenced by it. Had we been asked to write the program from scratch, chances are that we would have written one monolithic main class performing all the computation done by the program, both the user-interface and the loan processing; instead of, a main class that implements the user-interface and delegates loan processing to `ALoanPair` (which in turn delegates to `ALoan` and `AnotherLoan`). This decomposition of the program was, in some ways, forced on us by the `ObjectEditor`, because it is based on the principle that at least two classes should be involved in the running of a program, a class written by the programmer, and another, `ObjectEditor`, responsible for providing a main method that drives the programmer-defined class. When we decided to replace `ObjectEditor` with our own main class, we retained this basic decomposition, as shown in Figure 9.

In some respects, the monolithic class is easier to develop in that we do not have to bother defining the interfaces between the classes of a decomposed program. In particular, it does not require us to export properties from one class to another. However, as we saw in the previous chapter, program decomposition has long-term benefits, since it encourages the development of reusable code. For instance, as shown in Figure 9, we have used `ALoanPair` for implementing two different user-interfaces, the spreadsheet-based one implemented by `ObjectEditor` and the console-based one implemented by `ALoanPairDriver`. Thus, even when we develop programs from scratch, we should retain the decomposition forced by `ObjectEditor`, with the main class mainly serving as a driver for one or more other classes. The main challenge for us will be to identify as many such classes as possible to support a high degree of componentization.

⁴ In some languages such as Pascal, it can be a procedure that returns results through its parameters.



Library Class for Console Input

We can, in fact, further decompose our program into independent components. The method `readInt` is a general purpose method, and really belongs in a separate class that can be used by any class that needs to read an `int` value from the console window. In this class also belong methods to read values of other standard types such as `double` and `boolean`. Let us call this class, `Console`. Its code is given below:

```
import java.io.BufferedReader;
import java.io.InputStreamReader;

public class Console {
    static BufferedReader inputStream = new BufferedReader(
        new InputStreamReader(System.in));

    public static int readInt() {
        try {
            return Integer.parseInt(inputStream.readLine());
        } catch (Exception e) {
            System.out.println(e);
            return 0;
        }
    }

    public static double readDouble() {
        try {
            return Double.valueOf(
                inputStream.readLine()).doubleValue();
        } catch (Exception e) {
            System.out.println(e);
            return 0;
        }
    }
}
```

```

public static String readString() {
    try {
        return inputStream.readLine();
    } catch (Exception e) {
        System.out.println(e);
        return "";
    }
}
public static boolean readBoolean() {
    try {
        return new Boolean(
            inputStream.readLine()).booleanValue();
    } catch (Exception e) {
        System.out.println(e);
        return true;
    }
}
public static void pause() {
    try {
        System.in.read();
    } catch (Exception e) {
        System.out.println(e);
    }
}
}

```

Each of these methods parses the string the read by `inputStream.readLine` to a value of the appropriate type, using existing methods of existing Java classes for doing so and taking care of exceptions thrown by these methods. Since `ALoanPair` now no longer defines `readInt`, it must name the defining class when invoking the class method:

```

    Console.readInt();

```

Unlike the other classes we have written so far, this class is application-independent –it can be used to read Console input by any application. It is like some of the application-independent classes provided by Java that we have used, such as `Math` and `BufferedReader`. Such classes are called *library classes* or simply *libraries*.

By building this library class, we have further reduced the size of the main class and added components to our program. We will stop this process now, and later, when we look at the model-view-controller framework, identify additional ways to find reusable components.

Multi-Class Programs

Writing multi-class programs offers the same kinds of benefits as writing multi-method programs at the granularity of classes rather than the smaller granularity of methods. Thus, it allows us to understand, design, browse, and optimize each class independently. For instance, in this example, we can understand the class `Console` without worrying about or even knowing all the classes that use it.

Developing a multi-class program requires more work. As mentioned before, we must create each class in a separate file. For now, we will put all library classes in the same directory as the classes that use them⁵. A library class must be compiled before a class that uses it. Thus, in the example above, if we were to use the command window, we would execute:

```
javac Console.java
javac ALoanPairDriver.java
```

To run the program, as before, we simply need to name the main class:

```
java ALoanPairDriver
```

When the main class calls a method in some library, Java automatically finds it and dynamically links it with the main class.

Working with multi-class programs is much easier if we use an interactive programming environment such as Eclipse. In this case, all we have to do is add the library class to the project in which it is used - the build command compiles all project files in the right order.

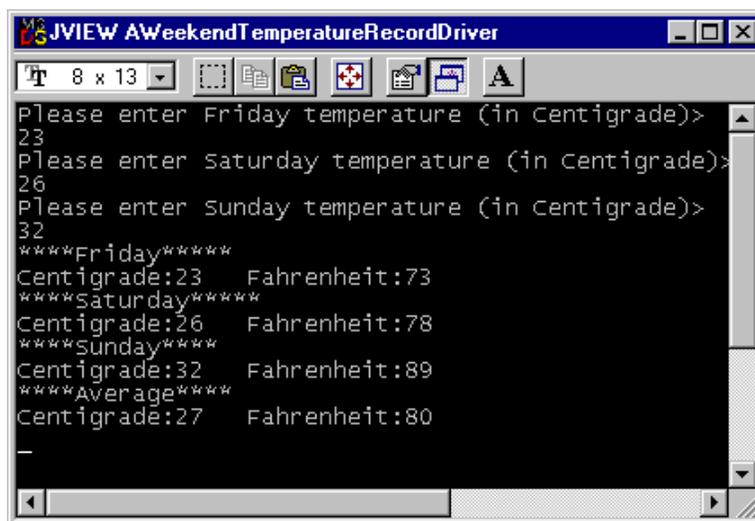
Summary

- A main method is a class method invoked by the Java interpreter to start a program. It must be named "main" and accept a String array as a parameter.
- The algorithm implemented by a complex method should have as many levels as possible.
- A program should have as many classes as possible. In particular, the main class should not be monolithic, that is, solve all aspects of the program. Like `ObjectEditor`, it should drive at least one other class.
- A method can throw an exception to indicate an error, which can later be caught by the caller of the method to report or recover from the error.
- Function calls should be used as expressions and procedure calls as statements.
- A function can have the side effect of displaying output, processing input, or changing global variables.
- A function should never change global variables.
- A debugger can be used to single-step through the various levels of the program.

⁵A library class in another directory can be accessed by listing the directory in the CLASSPATH.

Exercises

1. What are the advantages of tracing a program using a debugger rather than print statements?
2. What are the advantages of creating as many levels as possible in an algorithm?
3. What are the advantages of creating as many classes as possible in a program?
4. Implement a console-based program that stores and then displays the temperatures recorded during some weekend. It takes as input the temperatures, in centigrade, of the three days of the weekend, and shows each of these temperatures, together with their average, in both centigrade and Fahrenheit. You should use the classes you created as part of your solution to problems of the previous chapters. However, even though it is convenient to do so, do not use the class representing the weekend temperature. Instead use directly the two implementations of temperature. The reason is that this way your program will extend more easily to extensions of this problem given in the next two chapters.



```
JVIEW AWeekendTemperatureRecordDriver
Please enter Friday temperature (in Centigrade)>
23
Please enter Saturday temperature (in Centigrade)>
26
Please enter Sunday temperature (in Centigrade)>
32
****Friday*****
Centigrade:23 Fahrenheit:73
****Saturday*****
Centigrade:26 Fahrenheit:78
****Sunday*****
Centigrade:32 Fahrenheit:89
****Average*****
Centigrade:27 Fahrenheit:80
-
```

You can use the Keyboard class presented in this chapter.

5. Inspect the values of the local variables of the main method of this class before and after the user has input the temperatures. To show your instructor that you did so, print the local-variables window before and after the input. The easiest way to capture a Microsoft window is to select it, click on the Print Screen Keyboard button while holding down the Alt keyboard button. This puts the window in the clipboard. Now you can insert it into a Word or other kind of document, which can then be printed.

6. Consider the following class:

```
public class PureImpure {
    static int counter = 3;
    static public int getCounter() {
        return counter;
    }
    static public void setCounter(int newVal) {
        counter = newVal;
    }
    static public int getAndSetCounter(int newVal) {
        int oldVal = counter;
        counter = newVal;
        return oldVal;
    }
    static public int square (int arg) {
        return arg*arg;
    }
    static public void main (String[] args) {
        System.out.println(square (getCounter()));
        setCounter (3);
        System.out.println (getAndSetCounter(4));
        System.out.println (getAndSetCounter(4));
    }
}
```

- a) What are the local and global variables of `getAndSetCounter()`?
- b) Which of the functions are impure functions?
- c) Which of the impure functions have undesirable side effects?
- d) What is the output of the main method?