A Development Project

L3.1 A Guided Tour of an API

Follow these steps to explore the Java API.

- Launch your browser, and visit the Java API. Maximize the window, and take a close look at the structure of its contents. As we move closer to component-based programming, learning how to read and comprehend APIs will become the key programming skill.
- Notice that the screen is divided into three panes: (1) the upper-left pane lists the packages, (2) the pane below pane (1) lists the classes, and (3) the (large) pane to the right has descriptions. The three panes are "synchronized": if you click on a package in (1), only its classes will be listed in (2), and if you click on a class or a package in (2), only its description will be shown in (3). Spend some time familiarizing yourself with the navigation.
- Descriptions in (3) are available at the class level, package level, or all package levels. To choose, you click a class name, a package name, or the *Overview* link at the top, respectively.
- Click (in the upper-left pane) on the `java.lang` package so that its classes are listed in the lower-left pane. Click on the `Math` class, and examine its description in the right pane. It starts with an unstructured textual description of the class followed by two structures (each identified by a blue banner): *Field Summary* and *Method Summary*. These two blocks reappear afterward as *Field Detail* and *Method Detail*.
- The first field listed in the *Field Summary* is `E`. The left column describes this field as `static`. This, as we shall see, has a profound effect on how this field is accessed. If you do not see this word in the left column, then the field is not static. The left column also indicates that the type of this field is `double`. The type of data that a field can hold could be primitive or nonprimitive. Nonprimitive types, such as `String`, are in fact class names, and for that reason, their very first character is capitalized and their name is fully qualified, for example, `java.lang.String`.
- What can you say about the second field, `PI`? Is it `static`? What is its type?
- It is not a coincidence that all fields are `static`. In this lab we are focusing on classes with static features (utilities). Most classes are not like that, and we will turn to nonstatic fields and methods in the next lab.
- Look at the Method Summary. The method `random()`, for example, does not take parameters (because there is nothing between the parentheses), while `rint` (which appears after it) does take a parameter (a `double`).
- The name of a method together with the number, order, and types of its parameters is its *signature*. The signature of `rint` is `rint(double)`.
- It is OK for two or more methods in a class to have the same name as long as their signatures are different. In this case, we say that the method is *overloaded*. For example, the `max` method is heavily overloaded.

- The left column indicates whether the method is `static`. If there is no indication, then the method is not static. All methods in this class are static.
- Unlike fields, methods do *not* store data and, hence, do not have types. Some methods, however, return data and can therefore be indirectly associated with types. The `rint` method, for example, returns a `double`, which is why the API gives it this type in the left column. If a method does not return any data, the API describes it (in the left column) as `void`.

Static fields and methods are the easiest to use: all you need is to precede the name (of the field or method you want to use) by the name of the class and a dot. We say that we invoke them *on* the class. For example, the `E` field of `Math` can be accessed via

```
Math.E
```

Its content can be retrieved and stored in a local variable by writing

```
double x = Math.E;
```

Similarly for methods, to invoke a method such as pow, which computes the result of raising a number to another, you would write

```
Math.pow(2, 3);
```

This computes 2^3, or 8. The parameters you pass must be compatible with the types declared in the signature. Note that automatic promotion applies to parameters. The above example is correct even though there is no method with signature:

```
pow(int, int)
```

This is because the compiler will bind the above invocation with

```
pow(double, double)
```

L3.2 A Software Project

We will follow the formal software development process to approach our first project.

Analysis

In this phase, we need to determine *what* the project is all about and specify *exactly* its input and output. After meeting with the customer, our analyst determined that the sought system should be able to convert temperature from Fahrenheit to Celsius. The input must be prompted with the message

```
Enter the temperature in Fahrenheit
```

The entry is made on the next line; it is a whole number that is not less than –250. If the entry is not a whole number, an exception should occur. If the entry is not in range, a runtime error with the message *Value out of range* should occur. The output consists of one line containing, from left to right, the following elements separated by a space: the entered temperature,

letter "F," an equal sign, the corresponding Celsius temperature rounded to one decimal, and the letter "C." Here is a sample run of the proposed system:

```
Enter the temperature in Fahrenheit
55
55 F = 12.8 C
```

Design

In this phase, we make a plan for *how* the system will accomplish its goals by breaking the problem into tasks and identifying the classes and methods responsible for each. The tasks in our project are the following: printing to the screen, reading from the user, converting, and formatting. Before developing a method for each, a good Java designer starts by searching the market to see if a class containing some of the needed methods is already available. Indeed, the ready-made classes `Scanner` and `PrintStream` contain methods for reading input and for writing and formatting output. This leaves the conversion issue, and our physics textbook says that to convert a temperature f in Fahrenheit to a corresponding one c in Celsius, we use the following formula:

$$c = 5 (f - 32) / 9$$

Hence, all we need to develop ourselves is an app (`Lab3`) that invokes the needed services and performs the conversion.

Implementation

In this phase, we write and compile the `Lab3`. We need a local variable to hold the number entered by the user. Since this is a whole number, we will declare it to be of the `int` type. We must give it a name that describes its content (a temperature in degrees Fahrenheit) and that abides by our style guide. This leads us to the declaration

```
int tempF;
```

We also allocate a second variable to hold the temperature after the conversion. Since the formula yields a real number, we will declare it as `double`:

```
double tempC;
```

The conversion formula also involves the constants 5, 9, and 32. It is possible to incorporate them directly in a Java expression, without allocating variables for them, but that would violate our coding style, which dictates that no magic numbers (except 0, 1, and 2) may appear in expressions. Hence, we declare them like other variables, but we also state that they are constants by using the keyword `final`:

```
final int ZERO_SHIFT = -32;
final double SCALE_FACTOR = 5.0 / 9.0;
```

We gave each a meaningful name and type and used all caps (and an underscore to separate words) as per our coding style for constants. Using these variables, we can perform the conversion using the following assignment statement:

```
tempC = SCALE_FACTOR * (tempF + ZERO_SHIFT);
```

This leaves I/O, validation, and formatting. Complete the development of this application.

Testing

We will use *black box* testing to establish confidence in the correctness of our program. This means we pretend that we do not know how the program is written, only that it takes an input and produces an output. We supply various input cases to it, examine the output for each, and compare it with what we deem to be the correct answer. Here are our findings for valid test cases (ones that abide by the validation rules specified in the analysis):

- Normal cases: 32 and 212
- The program produced the correct answers.
- Extreme & boundary cases: –249 and –250
- The program produced the correct answers.

And here are our findings for test cases involving various invalid inputs:

- Out of range: –251 and –600
- The program produced the specified error message.
- Fractional: 7.5, –12.4, and 0.5
- The program generated a run-time error: `NumberFormatException`
- Non-numeric: testing, +32, and 2020CSE
- The program generated a run-time error: `NumberFormatException`

Exercises

Programming exercises are indicated with an asterisk.

3.1 Visit the Java API. (A link to it was added to your browser in Lab 2.) One of the classes is called `Color`. (a) In which package does this class reside? (b) Is this a utility class? (c) One of the fields in this class is called `"blue"`. Is this field final?

3.2 Visit the Java API. (A link to it was added to your browser in Lab 2.) (a) In which package does the `Currency` class reside? (b) Is this a utility class?
Hint: The absence of a constructor section is necessary but not sufficient for the class to be a utility. Check whether all the features are `static`.

3.3 Visit the Java API. (A link to it was added to your browser in Lab 2.) Two of the classes are called `Date`. (a) How can there be two classes with the same name? (b) If such a name were referenced in a program, how would the compiler know which one to bind with? (c) Is the `compareTo` method in `java.util.Date` overloaded?

3.4 Visit the Java API. (A link to it was added to your browser in Lab 2.) One of the classes is called `Double`. (a) In which package does this class reside? (b) Write a short program to output the values of its `MAX_VALUE` and `NaN` fields.

3.23 The web site for this book contains a collection of tests. You are now in position to take Test A, which covers Chapters 1 to 3. Follow these steps:

- Print and read the outline of the test. It tells you what kind of questions will be on the test, how they are weighted, and what aids are allowed.
- Print the test and take it. Do not use any book or API, and ensure that you do not exceed the allotted time.
- After taking the test, read the answers, and use their marking guidelines to mark your own test.

eCheck

Check03A (TS = 16)

Implement the project described in this lab using the app name `Check03A`. Here are more sample runs:

```
Enter the temperature in Fahrenheit
76
76 F = 24.4 C
Enter the temperature in Fahrenheit
-5
-5 F = -20.6 C
Enter the temperature in Fahrenheit
112
112 F = 44.4 C
Enter the temperature in Fahrenheit
212
212 F = 100.0 C
```

Once you are comfortable with the app behaviour, run it through eCheck.

Check03B (TS = 21)

Given the altitude of a satellite above Earth's surface, `Check03B` computes the satellite's orbital period, that is, the time it takes the satellite to make one complete revolution around Earth.

Requirements analysis revealed that your system should prompt for, and input on the same line, the altitude in kilometres (a whole number) without any validation. It should then output, on one line, the computed period in hours (whole number), minutes (whole number), and seconds (one decimal), exactly as shown in these three sample runs:

```
Enter the satellite altitude in km ... 1000
*******************************************
Orbital period = 1 hours, 45 minutes, and 5.7 seconds.
Enter the satellite altitude in km ... 35800
*******************************************
```