Programming in Java (23/24) – Day 3:
Simple and complex data types, more on branches

1 Introduction

Programs have lots of data, they are basically ways of getting data, transforming data, and giving data back to the world. Data types, as we have already seen, tell you which kinds of data you have in your program.

Computers store data in their memory in the form of bits. The word "bit" is a contraction of "BInary digiT", so a bit is either 1 or 0, true or false, high or low, on or off. Bits are organised in groups of 8 called *bytes*. These bytes —groups of eight bits— are used to store any data in the memory of the computer. When you have got 1024 bytes, you have got a kilobyte or kB; when you have 1024 kB you have got a megabyte or MB, and so on for gigabytes (GB), terabytes (TB), and petabytes (PB). Note that in computing everything is measured in powers of 2 ($2^3 = 8, 2^{10} = 1024$) and not in powers of 10 as in normal life (10, 100, 1000. . .). That is because computers count with bits (2) and human beings count with their fingers (10).[1]

2 Simple data types

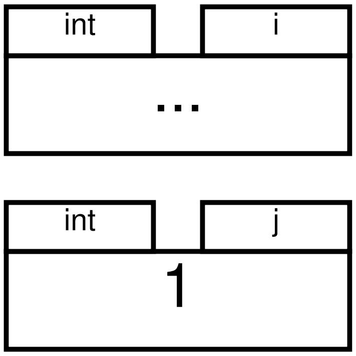
Simple data types can be thought as boxes in the computer's memory. Every time you declare a variable in your program, you can think[2] of the computer as creating a little box in its memory to store your variable. That box has two tags on it: one of them holds the name of the variable and the other holds the type (Figure 1). In the same way, you can think of assignment as putting a value inside that box (Figure 2).

2.1 Static typing and dynamic typing

You have already seen that Java, as most languages, puts some restrictions to what you can use as a "name" tag for your boxes. You know that they have to start with a letter: "count" is a valid name but "/me" or "5thNumber" are not. You also know that some words cannot be used by the programmer for variables names because it would be confusing for the computer, e.g., "while", "if", "int", or "boolean".

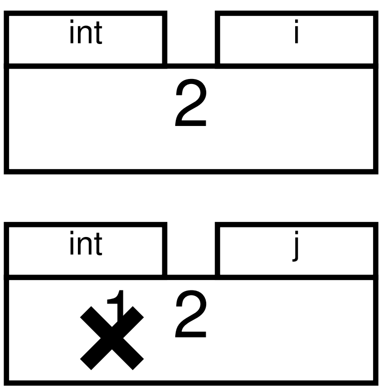
[1] Not all humans used only the ten fingers in their hands to count. Some languages like French or Irish have remains of base-20 counting.

[2] This is only a metaphor and is not supposed to be a fully accurate description of how memory is managed on a modern computer, of course.




```
int i;
int j = 1;
```

Figure 1: Declaring a variable can be seen as creating a box. The box has a tag for the name and another for the type.




```
i = 2;
j = 2;
```

Figure 2: Assigning a value to a variable can be seen as putting a value in the box. If there was something in the box, it is overwritten and lost forever.

Many programming languages also place one restriction on the "type" tag: once you decide the type of a variable, you cannot change it. This is like people having static opinions that they do not want to change. This type of languages are called *statically typed languages* and, contrary to people with fixed ideas, they are not necessarily bad or obnoxious. Java is an example of a statically typed language. Using statically typed languages is generally advantageous for developing large programs.

Some programming languages allow you to change the type of your variables as you go along, so you can have a variable that sometimes is an **int** and later in time is a **boolean** or a String. These languages are called *dynamically typed languages*. They have pros and cons compared to their statically typed counterparts. Python is an example of a dynamically typed language.

Since Java is a statically rather than a dynamically typed languages, it does not let us write the following:

```
int x = 4;
x = "Hello"; // type error! "Hello" is a String and not an int
```

Similarly, in Java a variable cannot be an **int** and then be declared again as a String:

```
int x = 4;
String x = "Hello"; // cannot declare x more than once!
```

There are many more things to know about typing in programming languages (strong vs. weak, inferred vs. manifest, duck typing, and much more) but for now it will not be necessary to go into those details.

2.2 Most common simple types

2.2.1 Integer numbers

The integer type **int** is probably the most used simple data type, as integer numbers are used for two of the most common operations in computing: counting and indexing. Integers use 32 bits (4 bytes) of memory, and an integer variable can hold values between -2,147,483,648 and 2,147,483,647 (inclusive). This data type is large enough for the numbers needed in 90% of programs most people write. We have already seen how to use it:

```
int count = 1;
```

There are three other kinds of integers for some special uses. When a program needs very large (positive or negative) values, there is a type **long**, for *long integer*. It is called "long" because it uses 64 bits instead of 32, which means a long integer variable can hold values between about $-9.22 \cdot 10^{18}$ and $9.22 \cdot 10^{18}$. There is also a "short" integer that uses only 16 bits of memory (**short**); this was sometimes useful to save memory when Java was first released in 1995 and computers were more limited but is hardly ever used with modern computers. Finally, there is a "byte", an integer number that uses only 8 bits of memory (**byte**). This data type can be useful for working with binary data, i.e., data that is not necessarily stored in text format, such as music or videos.

2.2.2 Floating-point (decimal/rational) numbers

Not all numbers are integer. Examples of common non-integer numbers include the result of a division of two integers where one is not a multiple of the other, and real-measurements like your height, your weight, and the distance between your workplace and your home (unless you work at home). In maths, these are called real numbers. In computing, they are usually called floating-point numbers and are represented as a list of significant numbers and an exponent. The term “floating-point” refers to the fact that the decimal point can "float", that is, it can be placed anywhere in the number as long as the the exponent is changed accordingly.

$$1.23 \cdot 10^{-3} = 123 \cdot 10^{-5} = 0.00123 \cdot 10^{0}$$

We cannot use superindex notation in a plain-text file, so we need a special way of writing these numbers. Those three ways of depicting the number above can be written as `1.23E-3`, `123E-5`, and `0.00123`, and the three are equivalent.

In Java, real numbers are usually represented with the simple data type **`double`**, that uses 64 bits. There is also a 32-bit version called **`float`** but, as with **`short`**, it is hardly ever used today.

Important note. Floating-point numbers do *not* have infinite precision, and operating with them can cause rounding errors. There is a special type for representing real values where precision is paramount (like in banking): `BigDecimal`.

Equality. Due to rounding errors, it does not make sense to test for equality among real numbers as we can do with integers. Two numbers could be notionally the same but be different due to rounding errors, so they are never compared with equality with “==”. Instead, what is usually done is test whether the difference is less that some precision limit appropriate for the application (e.g., 1.0E-6), as in the example below. In other words, it is checked whether the two numbers are “close enough”. Note: `Math.abs()` returns the absolute value of the number inside the parentheses, e.g., `Math.abs(-3)` returns 3.

```
        double d1 = (0.1+0.1)/0.3;
        double d2 = 2.0* 1000.0/9000.0*3.0;
        // WRONG!
        if (d1 == d2) {
           // this is not printed due due to rounding errors
           System.out.println("They are the same (wrong comparison)");
        }
        // RIGHT!
        if (Math.abs(d1 - d2) < 10E-6) {
           System.out.println("They are the same (right comparison)");
        }
```

2.2.3 Boolean (binary) values

The simple data type **boolean** represents one bit of information. It can hold the values **true** and **false**.

2.2.4 Characters

Text is composed of characters: 'a', 'b', 'c', . . . The **char** simple type is used to represent characters. It uses 16 bits, meaning it can represent any of 65,536 different characters.

Actually, the 16 bits of a **char** represent a Unicode symbol. Unicode is a computing industry standard for the consistent encoding, representation and handling of text expressed in most of the world's writing systems. It includes symbols from most writing systems in the world, including alphabets like Latin, Cyrillic, Arabic, or Hebrew; syllabaries like Japanese katakana and hiragana, or Cherokee; and many more.

Characters can be used alone in any Java program, can be printed on the screen, and can also be combined to form our well-known Strings, as in the following code:

```java
public class D3Example {
    public static void main(String[] args) {
        char firstChar='J';
        char secondChar='i';
        char thirdChar='m';
        String jimName = "" + firstChar + secondChar + thirdChar;
        System.out.println("Name: " + jimName);
        System.out.println("Letters: " + firstChar + "," + secondChar + "," + thirdChar);
    }
}
```

Listing 1: D3Example.java

Interestingly, **char** is also considered as an integer number in Java:

```java
        char c = 'a' + 1;
        System.out.println(c);
```

will print the letter b on the screen.

You may have noticed that we have not mentioned `String` among our simple types. This is because `String` is a *complex type*.

3 Complex types

Complex types are types of data that do not fit in a box, not even in one of the big 64-bit boxes used for **double** or **long**. Because they do not fit in the "boxes", computers have to store them somewhere else. However, they also need to know where they are... and that is what the boxes are used for.

Modern computers have *a lot* of memory. Long forgotten are the days when Bill Gates allegedly said: "640kB of memory should be enough for everything". Part of a computer's memory is used for the boxes (in a part of memory called "the stack") and most of the rest is used for everything else, including complex data (that part is called, quite unceremoniously, "the heap").

When your Java code uses some complex data, the computer stores that data in some region of the heap —identified by a *memory address*; then it stores the address in a box in the stack, much in the way it stores integers and booleans. This looks similar to Figure 3. The memory address in the box can be seen as "pointing to" the place in memory where the real data is stored. For this reason, we will call it a *pointer*.

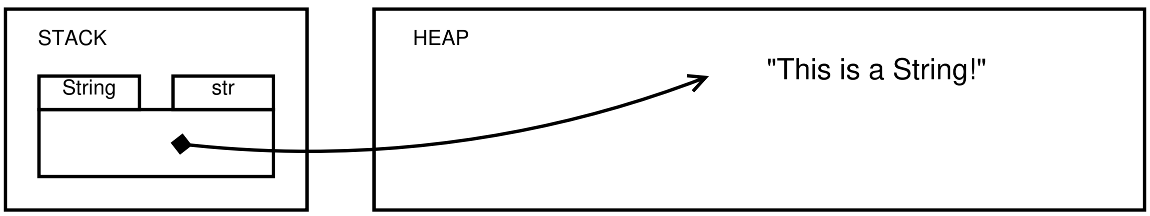


Figure 3: A String is a type of complex data type. The data itself is stored in the heap and its address is stored in box in the stack —pointing to it.

Using complex data types is different from using simple types in several ways. For starters, as complex types are not stored in the stack but in the heap, you have to *allocate* memory-space in the heap to store the data. This is done with the reserved word **new**; you also need parentheses to pass arguments (as if using a method: this is because complex types have something called a *constructor* that we will study in more detail later). We are now going to see several examples, starting with the pervasive Strings.

One final note about names. Complex data types usually have names that start with capital letters. This is not compulsory but it is what everybody expects: simple types with non-capital letters (e.g., **int**, **double**) and complex types with capital letters (e.g., `String`, `List`, `Scanner`, `Customer`, ...). If you do not capitalise your complex types, other Java programmers will get very confused when they read your code. This is not only impolite, it is also unprofessional.

3.1 Declaration and initialisation

When you are working with simple types, the memory is always used as soon as you declare the variable. In other words, your program will use the same amount of memory if you type `int i;` and if you type `int i = 1;`: it always uses 32 bits of your computer's memory.

This is not true with complex types. Declaring a variable with a complex type (e.g., `String str;`) only makes the computer reserve the box for the pointer, but nothing else. It is only when the memory is *allocated* (e.g., `str = new String();` or `str = new String("This is a String");`) that the total amount of memory for the data is used. Note that the difference is huge. Boxes are at most 64 bits big, but there is no limit to the size of a complex type: it could be several kB or even MB.

Sometimes the pointer will not point to any address in memory. This can be useful in some cases that will become clearer as we learn more about programming, including error detection and release of memory that is no longer needed (remember that complex types can use a lot of memory). To make a pointer point to nowhere, we use the reserved word `null` (with non-capital letters).

```
String str;
str = null;
```

A pointer pointing to null is called a *null pointer*. This basically means that the address in the box is zero (rather than an obscure hexadecimal number like 0x1a3ec74); the computer knows there is never anything at address zero, so it knows a null pointer is not being used to point to any real data (Figure 4).

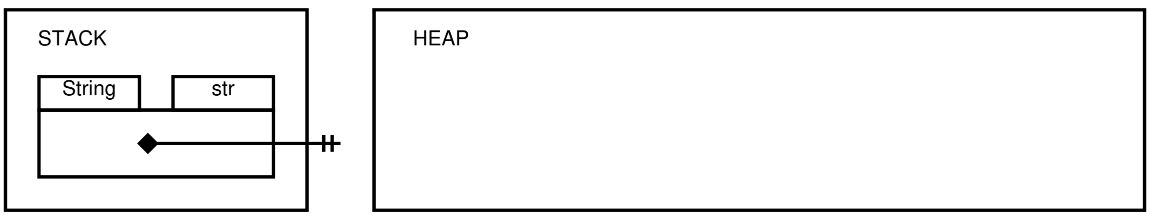


Figure 4: A null pointer is basically a zero address, pointing nowhere.

As a null pointer does not point to any real data, if you try to access a variable that is pointing to null, the computer will complain with a `NullPointerException`, as in the following example (try it!):

```
String str = null;
System.out.println(str.length());
```

3.2 String

Strings are everywhere. Every program, except the most trivial, uses strings: user names, passwords, addresses, configuration options, data input from the keyboard, a web page read through the Wi-Fi connection... almost anything is a `String`. It is the most widely used complex type.

At their most basic, strings are sequences of characters. You already know how to read a piece of text from the user:

```
java.util.Scanner scan = new java.util.Scanner(System.in);
String str = scan.nextLine();
```

If you want to create a string in your program without reading it from the user, the basic form of creating a string is like this:

```java
        String str = new String("This is a String");
```

but you already know that this can be done in an easier way:

```java
        String str = "This is a String";
```

The former two statements do essentially the same job;[3] only the second is more convenient and most people use it instead of the other.

Notice that single quotes are used for characters and double quotes are used for strings: `'c'` is the character *c*, but `"c"` is a String with only the character *c* in it. So, it is important to note that in Java a `String` with only one character is still a `String`.

You already know that you can do some things with strings using *methods*. Although we are going to learn more about methods in the next days, we can introduce some of them now to help with the exercises. Assuming you have a `String` called `str`, you can use the following methods:

str.length(): this method returns the length of the string.

str.charAt(): this method requires an integer inside the brackets, and returns the character at the specified position; note that the first character is at position 0, not 1.

str.substring(): this method requires two integers inside the brackets, separated by a comma, and returns another string that begins at the character specified by the first number and extends to the character specified by the second number minus one[4].

Here is a brief example of how they are used. Note that if you want to use double quotes inside double quotes, you have to *escape* them using the backslash ("\") sign: the backslash will make Java treat the next double quote as a literal double quote and not as the end of the string.

```java
        String str;
        str = new String("This is an example");
        System.out.println("Initial string: \"" + str + "\""); // Note the escaped quotes!
        int l = str.length();
        System.out.println("The length of the string is " + l);
        char c = str.charAt(0);
        System.out.println("The first character is " + c);
        String str2 = str.substring(8,18);
        System.out.println("The substring from char 8 to char 17 is \"" + str2 + "\"");
```

The result of this little program is:

```
Initial string: "This is an example"
The length of the string is 18
The first character is T
The substring from char 8 to char 17 is "an example"
```

[3] There is a subtle difference, and we will learn more about it when the time comes.

[4] This results in the substring having a length of secondNumber - firstNumber.

3.3 You own structures: classes

`String` is the most common complex type, but it is not the only one. As a matter of fact, programmers can create their own complex types very easily. In order to create new types of complex data, we use the keyword **class**. You can think of it as creating new classes of data.

A new class of data must have a name (remember: by convention, complex data types start with capital letters, classes too) and be defined in between curly brackets. A complex type is composed of several other types, simple or complex. Let's see an example, which we would store in a file `Person.java`:

```java
class Person {
    String name;
    int age;
}
```

This code defines a new class of data that represents a person, and it is composed of a string for the name and an integer for the age. As you can see, both simple and complex types can be used inside a class. You can even have data of a specific class inside the same class, as in the extended example below:

```java
class Person {
    String name;
    int age;
    Person parent1;
    Person parent2;
}
```

Now a person has a string for a name, an integer for the age, and two variables of type `Person` for the two parents. Once you start working with complex data on a regular basis (which means: from now on) you can see that it is crucial to use good identifiers for your names. Compare the previous class with this very-badly-named example, which would be stored in a file `P.java`:

```java
class P {
    String s;
    int n;
    P p1;
    P p2;
}
```

Both examples contain the same information at a logical level, and the computer will behave equally well with both. However, human programmers reading the code will have a very hard time trying to understand what the second class really represents, while the first example is obvious. Remember: **source code is written once but is read many times**, so your code should be as clear as possible.

Variables in a class are often called *fields*. Sometimes they are called *member fields* because they can be seen as being "part of" (members of) the class. So the class `Person` can be said to have four fields: one of type `String`, one of type **int**, and two of type `Person`. Fields in a class are accessed using a dot, as in the following example (which we would write not in class `Person`, but in a file with some other name, say `MyEmployeeProgram.java`, in its `main` method, as usual):

```java
Person employee = new Person();
employee.name = "John Smith";
employee.age = 45;
System.out.println("BOSS: How old are you, " + employee.name + "?");
System.out.println("EMPLOYEE:  I am " + employee.age + " years old.");
```

In this example, we have created one variable of class `Person`, an *instance* of the class. Each of these instances of a class are also called *objects*.

Every time you use the keyword **new**, you are giving the computer an instruction to create another object (this involves looking for a place in memory to store it and placing the corresponding pointer pointing to it). You can think of a *class* as a blueprint for objects, and thus creating a new instance (by using **new**) is creating a new object according to that blueprint.

Note that fields on an object are accessed using a dot to separate the name of the variable that identifies the object (e.g., `employee`) and the name of the variable inside the object, the field (e.g., `name`), as in `employee.name`.

We can make the example slightly more complicated by using more than one instance of the class `Person`.

```java
Person john = new Person();
john.name = "John Smith";
john.age = 35;
Person mary = new Person();
mary.name = "Mary Smith";
mary.age = 32;
Person student = new Person();
student.name = "John Smith, Jr.";
student.age = 5;
student.parent1 = john;
student.parent2 = mary;
System.out.println("TEACHER: How old are you, " + student.name + "?");
System.out.println("LITTLE JOHN: I am " + student.age + " years old, sir.");
System.out.println("TEACHER: Who are your parents?)");
System.out.println("LITTLE JOHN: " + student.parent1.name + " and "
                                   + student.parent2.name + ", sir.");
```

There are three important things to learn from this example.

First, every instance of a class has its own copies of its fields. This is why we can have two instances of class `Person` in the computer's memory at the same time where the same fields have different values (here, e.g., `age` with `35` and `32` in the first two objects that we have created). Thus, instead of the term *fields*, we will often also use the term *instance variables*, to highlight that each instance has its own copies of these variables.

Second, we can access data inside an object that is inside another object. Actually, there is no limit to the level of nesting you can achieve with complex data (as long as you do not run out of memory). If you want to access a field of a field you only need to use two dots, as in `student.parent1.name`.

Third, we have not initialised all fields in the objects of class `Person`. When you create a new object, the fields of the object have to be initialised as any other variable, otherwise they will not have the values you want them to have. In this example, we do not know who the respective two parents of objects `john` and `mary` are. If we try to access them, the computer will complain. On the other hand, if `parent2` of object `mary` was properly initialised, we could know the name of a grandparent of the student by typing something like:

```
System.out.println("The grandparent's name is " + student.parent2.parent1.name);
```

In the next day, we will see how to add methods to your classes, like those that you know from `String`. This will also allow us to avoid having to access the instance variables of other objects directly.

3.4 Boxed types: Integer, Double, and Character

For every simple type in Java, there is a "boxed" complex version. The most important ones are summarised in Table 1. Note that the name of the complex type uses capital letters and is sometimes a longer, complete-word version of the simple type's name.

Simple	Complex
int	Integer
double	Double
char	Character

Table 1: Most important simple types and their boxed counterparts

The complex version of a simple type works mostly like any other complex type. The variable itself contains a pointer that points to the address in memory where the actual value is stored (Figure 5). Why have complex versions of simple types? Other data types have a lot of information (i.e., how long is a `String`?) and that is why they need to be stored indirectly, but this is not the case with boxed types. Why then? Remember that complex types of data can have their own methods, and sometimes this is very useful. You have already come across one of the most used ones: `Integer.parseInt()`, which is used to convert strings that contain only a number into an **`int`**. Boxed types have several other methods, and we will see some of them, but `parseInt()` is the one most frequently used.

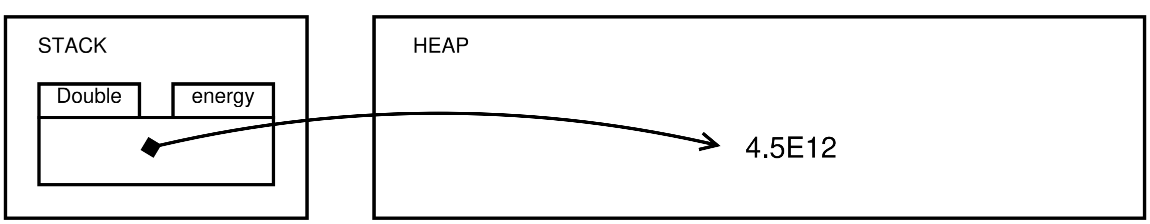


Figure 5: A boxed type, like any complex type, consists of a pointer to a place in memory where the actual data is stored

Why are these complex types called "boxed"? This is because they can be seen as a box wrapping around a simple type. In the old days, using boxed types was cumbersome and boring. You could need to type something like:

```java
        int i = 1;
        Integer boxedI = new Integer(i);
        int i2 = boxedI.intValue();
```

You had to explicitly create an instance of the boxed type like any other complex type by using **new**, and then you needed a method of the boxed type to get the value of the simple type from inside. This was a lot of work when you had many variables and had to perform conversion from simple type to boxed type very often, for example if you have to make comparisons of data between two different sources (one simple type, the other boxed type). This why Java now allows for what is called *auto-boxing*, which is just a fancy name to say that you can use them in exactly the same way and the computer takes care of all the boxing-in and boxing-out. In other words, the following code is perfectly legal:

```java
        int i1 = 8;
        Integer i2 = 7;
        if (i1 >= i2) {
            System.out.println("Nowadays you can use boxed and simple types together!");
        }
```

Note that you do not need to write explicitly `i2 = new Integer(1);`, but `i2` is still of a complex boxed type, its box contains just a pointer to some place in memory where a "1" is stored. The variable `i1`, on the other hand, is a *bona fide* integer variable, a box with a "1" inside (see Figure 6). However, thanks to auto-boxing, you can compare one with the other: the computer will automatically extract the "1" inside the boxed type to perform the comparison with the other "1". You do not need to do it yourself!

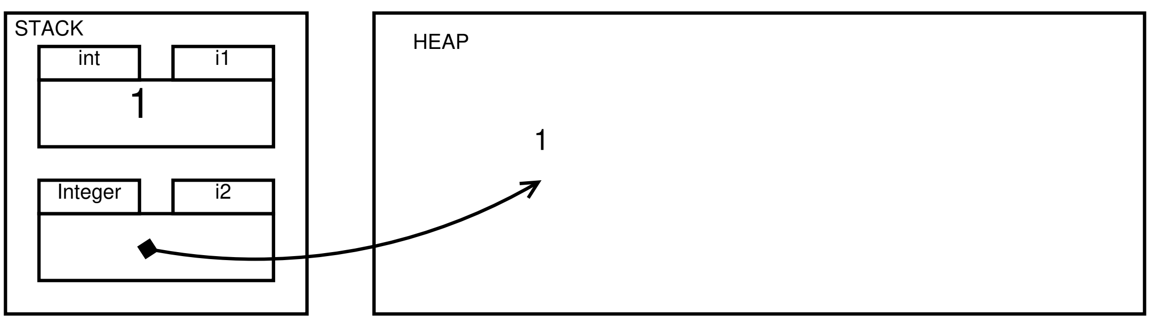


Figure 6: Even if `i1` and `i2` are, strictly speaking, two very different things, auto-boxing makes it easy to work with both types at the same time.

You may now ask: if boxed types are so useful, why do we need their simple versions as well? The answer is *efficiency*: whenever we create an instance of a complex type on the heap using **new**, the running Java program must spend some time on finding and reserving memory on the heap and putting a pointer to that memory into a box. In contrast, data that has a primitive type can be stored directly in a box, without the detour via the heap. Values of primitive types can also be processed very efficiently by the hardware of modern computers because they are so small and modern processors have built-in hardware support to execute instructions on them very quickly.[5]

3.5 A final note on terminology

If you read other books or web pages about Java, you may notice that they use terms that are different from the ones we have used in this section. We go through some of them here for the sake of clarity.

Simple type: A type that is stored in its own box, usually 32 bits or 64 bits long. These are usually called *primitive* types in the Java world. I prefer to call them simple data types here to distinguish them clearly from complex data types.

Complex type: A type that has two parts: the box contains a pointer, and it points to a place in memory where the actual value is stored. In the Java world, these types are simply called *classes* and the actual values are called *objects*, which is precise but fails to be explicit on the difference between simple and complex data types, and this can be a source of confusion.

Pointer: The content of the box in a complex type, a memory address where actual data is stored. This is sometimes called a *reference* or a *handle* to prevent confusion with pointers of other languages (like C) that behave in a slightly different way. I think many programming languages have constructs that share the same name and behave in slightly different ways (Strings are a major example) so this is not of much concern. Besides, it looks very confusing to me to access a null "reference" or a null "handle" and get a Null**Pointer**Exception as a consequence.

[5] On the MSc Computer Science, you will learn more about this topic in the modules *Fundamentals of Computing* and *Computer Systems*.

4 More on branches

You can write a lot of different programs with the little **if**...**else** and **while** constructs you have already learned and practiced, for executing different branches in your program and for running loops. There are, however, other constructs that you can use to make your programs clearer. In this section, we will learn about the **switch**...**case** multiple branching feature and the ternary operator (also called conditional operator). On later days, we will also see how to use the very useful **for** and *for each* loops.

4.1 Multiple branching

Look at the following code, which loosely resembles an automatic phone-answering program:

```java
public class D3Example1 {
    public static void main(String[] args) {
        java.util.Scanner scan = new java.util.Scanner(System.in);
        System.out.println("Please choose an option:");
        System.out.println("For 'Checking you balance', please enter 1");
        System.out.println("For 'Purchases', please enter 2");
        System.out.println("For 'Refunds', please enter 3");
        System.out.println("For 'Queries about the warranty', please enter 4");
        System.out.println("For 'Returning faulty goods', please enter 5");
        System.out.println("For any other query, please enter 0");
        int choice = scan.nextInt();
        if (choice == 1) {
            // go and check balance
        } else if (choice == 2) {
            // go and purchase something
        } else if (choice == 3) {
            // go and process refunds
        } else if (choice == 4) {
            // go and answer queries
        } else if (choice == 5) {
            // return faulty goods
        } else {
            // go and talk with a human operator
        }
    }
}
```

Listing 2: D3Example1.java

Such a long list of **if** branches can be a bit confusing for future readers. When there are many options, and especially if the options are finite and known in advance, it is better to make the multiple branching clearer by using a **switch**...**case** construct, as in the following example:

```java
public class D3Example2 {
    public static void main(String[] args) {
        java.util.Scanner scan = new java.util.Scanner(System.in);
        System.out.println("Please choose an option:");
        System.out.println("For ’Checking you balance’, please enter 1”);
        System.out.println("For ’Purchases’, please enter 2”);
        System.out.println("For ’Refunds’, please enter 3”);
        System.out.println("For ’Queries about the warranty’, please enter 4”);
        System.out.println("For ’Returning faulty goods’, please enter 5”);
        System.out.println("For any other query, please enter 0”);
        int choice = scan.nextInt();
        switch (choice) {
        case 1:
            // go and check balance
            break;
        case 2:
            // go and purchase something
            break;
        case 3:
            // go and process refunds
            break;
        case 4:
            // go and answer queries
            break;
        case 5:
            // return faulty goods
            break;
        default:
            // go and talk with a human operator
            break;
        }
    }
}
```

Listing 3: D3Example2.java

As you can see, the multiple branches are introduced by the keyword **switch** followed by some parameter, choice in this case. Then several possible cases follow, each of them starting by a semicolon. The whole list of cases is inside two curly brackets, but the code of each case does not need curly brackets.

There are two important things to note in this program. First, there is a keyword **break** at the end of each case. This is very important. Without it, Java will finish the execution of one case and continue with the next one. Second, there is a **default** case that is executed if none of the other cases matches. This is also important: always write a default case for your **switch** ... **case** structures. They are good for detecting errors in your program.

Exercise A

Write a program like the one in the last example, but add a little code for each case (some `System.out.println` statements will do) and remove the **break** keywords. Execute the program and see what happens.

Exercise B

Write a program similar to the one in the former example, but a bit better. Make the computer ask for an option and then execute the corresponding code in a **switch** ... **case** structure. If the user introduces an invalid option (e.g., 9 in the example above), make the computer say "That is not a valid option, please try again", and ask again until the user enters a valid numbers. *Hint:* You will need to use a loop and change the default case.

4.1.1 What types can go inside a switch

You may now wonder: is it only integers that we can use as the parameter of a **switch** statement? The bad news is that we cannot use arbitrary type of data, but only things like integers, characters (also known as *chars*), strings, and enumerated types (also known as *enums*, we will come to them in a second). The good news is that this is usually enough.

4.1.2 What are enums

Enumerated types (usually called *enums*) are just lists of *tags* that can be useful to make your program more legible. They can be used to specify constants that have some special meaning. Defining an *enum* in Java is really easy:

```
enum Day {
    MONDAY,
    TUESDAY,
    WEDNESDAY,
    THURSDAY,
    FRIDAY,
    SATURDAY,
    SUNDAY,
}
```

This code would go into a file `Day.java`. *Enums* are another example of complex types in Java.

Once an *enum* is defined, it can be used to declare new variables in the same way as for other types, like `String` or **int**.

```
        Day dayOfWeek;
        dayOfWeek = Day.MONDAY;
```

Now, an enum can be used in **switch** (or **if**) statements. Compare the clarity of this code:

```
        switch (day) {
        case 1:
            // do something here for this day
            break;
        case 3:
            // do something here for this day
            break;
        ...
        }
```

and this code:

```
        switch (day) {
        case MONDAY:
            // do something here for this day
            break;
        case WEDNESDAY:
            // do something here for this day
            break;
        ...
        }
```

In the second case, it is clear what each case is for. In the first piece of code, you need to guess: look at the code for each case, look at the surrounding code...it is harder. Programming is already a hard (and fun) activity. It is wise not to make it harder than it is; it does not make it any funnier.

There is a general lesson to learn here. What you see in the code above is an example of *magic numbers*: numbers that appear in your code and have an effect, but is difficult to understand what their effect is or why they are what they are (i.e. why a 1 and not a 5?). **Magic numbers make code harder to read and understand. Never use magic numbers in your code.**

4.2 The ternary operator (or conditional operator)

There is another form of writing a branch. It is not used as commonly, but you should know it exists. It is the so-called *ternary operator* (or *conditional operator*).

Most operators in Java are either *unary* (they take one argument, like logical NOT "!") or *binary* (they take two arguments, like logical AND "&&" or addition "+"). There is only one ternary operator, with three arguments, that is really another fancy way of writing an **if**...**else** clause. We will illustrate how it works with a simple example:

```
        java.util.Scanner scan = new java.util.Scanner(System.in);
        System.out.println("Enter a number: ");
        int i = scan.nextInt();
        String s = (i > 5) ? "Greater than 5" : "Not greater than 5";
        System.out.println(s);
```

This code behaves exactly in the same as the following code:

```
        java.util.Scanner scan = new java.util.Scanner(System.in);
        System.out.println("Enter a number: ");
        int i = scan.nextInt();
        String s;
        if (i > 5) {
            s = "Greater than 5";
        } else {
            s = "Not greater than 5";
        }
        System.out.println(s);
```

As you see, the ternary operator takes three arguments: a boolean expression and two values. If the boolean expression is true, then the second argument is returned; if it is false, it returns the third argument.

In some cases the ternary operator can make your code clearer, but these cases are few and far between. In general, it is easier to write *bona fide* **if** statements. However, you may find the ternary operator in code written by others, and it is thus important to understand how it works.

Exercise D

Write a program that reads a series of numbers from the user, until the user enters the String "END". The program should then print how many numbers were positive, negative, or zero. Write the program using (a) **if**...**else** clauses first and then using (b) the ternary operator. Is there any difference? In which case it is easier to write the program? In which case does it look clearer to you?

4.3 Increment operations

The ternary operator is not the only strange way of making your code shorter. Increment operations are so common that combinations of symbols are used —in a slightly cryptic way— to make those code lines shorter.

Increment lines are lines in which the value of a variable (typically **int**) is incremented by some explicit value or the value of another variable, for example:

```
        time = time + lap;
        depth = depth + drillDepth;
```

This can be written in a shorter way like:

```
        time += lap;
        depth += drillDepth;
```

The same principle can be applied to most binary operators, including subtraction, multiplication, division, and string concatenation; so:

```
record = record - difference;
result = result * 2;
name = name + newTitle;
```

become:

```
record -= difference;
result *= 2;
name += newTitle;
```

A particular example is the increment of integer variable by one or minus, used for all sorts of counters:

```
time = time + 1;
count = count + 1;
i = i + 1;
timesLeft = timesLeft - 1;
```

which are most often written much more shortly as:

```
time++;
++count;
i++;
timesLeft--;
```

There is a small difference between the line `i++` and the line `++i`. The former evaluates the statement and then increases the value of `i` while the later increases the value of `i` and then evaluates the statement. This a subtle difference that you should not worry about and should actually avoid in modern code. In other words: never write any code that would behave differently if you changes all instances of `i++` for `++i`. This is very easy to do by just following the following rule:

If you use i++ or similar, make sure it is the only thing you write in that statement; do not combine it with anything else.