Programming in Java (23/24) – Day 5: Constructors, access levels, arrays, static data, casts

1 Constructors

There are two important aspects of classes that we have to learn about. I am talking about constructors and levels of access.

When classes have been used in the preceding sections, they were used in two steps: first the memory was reserved for them using **new** and then their fields were initialised.

```
        Point corner = new Point();
        corner.x = 4;
        corner.y = 0;
```

This is not too cumbersome unless you have a class that has much more than two fields that need initialising. For example, if you wanted to create a Person with first name, family name, birth location, age, job, nationality, etc, you would need a lot of code every time you created a new Person.

```
        Person john = new Person();
        john.firstName = "John";
        john.familyName = "Smith";
        john.birthLocation = "London";
        john.age = 30;
        // the rest of the parameters would come here...
        // ...
        Person mary = new Person();
        mary.firstName = "Mary";
        mary.familyName = "Jordan";
        mary.birthLocation = "Edinburgh";
        mary.age = 33;
        // the rest of the parameters would come here...
        // ...
```

We have written a lot of code, and all that was just to create two objects. This is really boring. On top of that, it looks like we are repeating code over and over again by having to initialise all fields manually. There is a better way of doing this.

Every class can have one or more *constructors* (sometimes called *constructor methods*). A constructor is a special kind of method that is used to initialise an object of a class when it is first created, and it is executed every time a new object is created using **new**. In other words, the constructor is a way of telling Java to use **new** *both* to allocate the memory *and* to initialise the fields of the object at the same time. Simpler. Clearer.

A constructor looks like a method without a return type (not even **void**). The name of a constructor is the same as the name of the class for which we write it. Have a look at this example:

```
class Point {
    int x;
    int y;

    Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

    void moveTo(Point remote) {
        this.x = remote.x;
        this.y = remote.y;
    }

    // more methods here...
}
```

Elsewhere (e.g., in the main method of a class PointTester):

```
        Point point = new Point(1,1);
        System.out.println("The point is now at " + point.x + "," + point.y);
        Point remotePoint = new Point(10,20);
        point.moveTo(remotePoint);
        System.out.println("The point is now at " + point.x + "," + point.y);
```

No need to initialise the fields of the points after creating them. The constructor does it for both of them.

If a class has more than one constructor, Java chooses the right one according to the types of the actual parameters used for calling the constructor. For example, we could create a class Rectangle with two different constructors and then create instances (i.e., objects) of it using one or the other constructor:

```java
class Rectangle {
    int length;
    int width;

    Rectangle(int length, int width) {
        this.length = length;
        this.width  = width;
    }

    // This constructor creates a square, all sides equal
    Rectangle(int length) {
        this.length = length;
        this.width  = length;
    }
}
```

Elsewhere:

```java
        Rectangle dominoRectangle = new Rectangle(1,2);
        Rectangle pythagoreanRectangle = new Rectangle(3,4);
        Rectangle goldenRectangle = new Rectangle(1618,1000);
        Rectangle square = new Rectangle(5);
```

Here our two constructors look very similar internally. To avoid repeated code following the DRY principle, we would like to reuse an existing constructor when we write a second (or third or...) constructor. But so far we have seen calls to constructors only together with the `new` keyword, as in `new Rectangle(1,2)`. If we wrote `new` inside a constructor, we would create a second object. This is not what we want. That's why Java has a special notation for the call of one constructor to another, reusing the `this` keyword:

```java
class Rectangle {
    int length;
    int width;

    Rectangle(int length, int width) {
        this.length = length;
        this.width  = width;
    }

    // This constructor creates a square, all sides equal
    Rectangle(int length) {
        this(length, length);
    }
}
```

Here the second constructor calls the first constructor in via `this(length, length);` in line 12 with the information what values to use for running the constructor in line 5. After the constructor call in line 12, we could still add further statements to our second constructor in line 13 if we wanted. But if we call another constructor via `this(...);` to do part (or all) of the work of initialising the object's fields for us, that call must be right at the beginning of our constructor, and we cannot put other statements before the call to `this(...);` in our constructor.

Every class has **at least one** constructor. If it is not explicit —as it has been the case with all classes in the previous sections—, Java implicitly adds an empty constructor: a constructor with no parameters that does not initialise any field.[1]

```
    // If there is no constructor for Rectangle, Java
    // will add an "implicit empty constructor" like this
    public Rectangle() {
    }
```

2 Levels of access (`public`, `private`) and information hiding

A real programming project can have thousands of classes, and millions of live objects in memory at any given time (remember that we create a new object every time we use `new`). Keeping track of all the interactions of these objects becomes very complex very quickly. That is why programmers make an effort to keep the complexity as low as possible. One strategy to achieve that objective is to *hide* as much information as possible and leave visible only what is strictly necessary.

Imagine that you want to find out the total population of the European Union and you request data from every country. Governments of each country could give you access to their data stores about their residents to inform your calculations. These data stores may well be organised differently from country to country, depending on how their internal information systems are set up – country A may keep track of this information on municipal level, whereas country B may have a central data storage with information on all its residents.

Alternatively, they could deny you the direct access to their internal data storages, but instead each tell you their respective total population figure, without giving you all the details of how they calculated their figure.

It is clear that the second option makes your life much easier and at the same time allows the countries' governments to stay flexible with the way they represent information internally – when you come back next year to ask the same question, they may well have changed how they internally keep track of the information about residents, and your old approach to extracting the information from their data stores may no longer work.[2] However, they will still be able to give you an answer to the question "What is the size of your population?", which they may now determine a bit differently internally (for example, country A may have moved the information about residents from municipal to county level).

[1]Strictly speaking, the constructor is not empty: it calls the constructor of the parent class. We will see this when we learn about inheritance.

[2]Also from a data protection perspective, the second option is much better: information about individuals should be shared very restrictively.

This is the principle of **information hiding**.

Information hiding is achieved in Java by means of two[3] new keywords: `public` and `private`. The keyword `private` specifies that the field or method that comes after it will be visible only inside its own class. This should be your default choice as a programmer, especially for fields. The keyword `public` specifies that the field or methods that comes after it will be visible from everywhere, inside and also outside the class.

Classes can also be `public` or `private`. In the same way that `public` fields and methods are visible outside of their own class, a `public` class is visible outside of its file, and vice versa.

Rules of thumb for access levels

Over many decades, programmers have developed some rules of thumb to know what should be `public` and what should be `private`. The following rules cover 90% of all cases, but do not expect them to be valid in absolutely every situation.

- If a class has methods, its fields must be `private`. This is the most common case. (Making all fields `private` is often called *encapsulation*.)
- If-and-only-if a class does not have any methods (not counting constructors), its fields must be `public`.
- Methods of a class must be `private` unless their purpose is to be called from outside the class, in which case they must be `public`.
- Constructors must be `public`.

How would class `Point` be if we were doing things right and using `public` and `private` to specify the visibility of everything? Like this:

```java
public class Point {
    private int x;
    private int y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public int getX() {
        return x;
    }

    public int getY() {
```

[3]There is a third keyword called `protected` but it is less important and we will come to it when we talk about class hierarchies. As we have seen earlier, we can also omit the keyword for the visibility as a fourth option – this is called *default* or *package-private* visibility. Both `protected` and no keyword for visibility lead to a visibility level "in between" `public` and `private` , which shares disadvantages of these visibility levels. This is why `public` and `private` are the most commonly used visibility levels in Java.

```
        return y;
    }

    public void moveTo(Point remote) {
        this.x = remote.x;
        this.y = remote.y;
    }
    // more methods here...
}
```

Elsewhere:

```
        Point point = new Point(1,1);
        System.out.println("The point is now at " + point.getX() + "," + point.getY());
        Point remotePoint = new Point(10,20);
        point.moveTo(remotePoint);
        System.out.println("The point is now at " + point.getX() + "," + point.getY());
```

This is a bit more complex at first sight, but now we can understand all its components. What can we see?

- The class is **`public`**, because we probably want it to be used from other files. This is the usual case.
- The constructor is **`public`**. If it was not, it would be difficult to create new objects of type `Point` using **`new`** from outside class `Point`. Actually, it would be impossible.
- The methods are all **`public`** (at least the methods we can see in the example), because all of them should be available for use by other code that uses objects of type `Point`.
- The fields `x` and `y` are **`private`**. Fields should be **`private`** and be accessed through methods in 99% of all programs.[4]
- As fields are **`private`**, two *getters* or *accessor methods* have been added. If we want to be able to modify the individual coordinates of objects of type `Point` after they are created, we will need to add *setters* too.

3 Everything is a class

In Java, *all* the code comes inside a class.[5] Usually every class is defined in its own file, e.g., a `Person` class is defined in a file called `Person.java`. When a file contains only one class, this class must be **`public`** and have the same name as the file (without the `.java` extension).

[4] We will soon talk about *why* it is a good idea to keep other programmers from accessing fields in our classes directly. Our analogy with the population count of different countries already gives a first hint: if you allow other programmers direct access to your data, you can no longer change your representation of the data without creating problems. If you provide access only through methods, you can still modify the implementation of your methods in the next release of your software, where you have changed which fields you are using.

[5] Enumerated types with the keyword **`enum`** are a special kind of class.

Recall that before you can run a Java program, you must first *compile* it. This means that you pass it through a piece of software called a *compiler*. The compiler checks whether your program is acceptable according to the syntax of Java. If it isn't, the compiler issues one or more error messages telling you what it objects to in your program and where the problem lies. You try to see what the problem is, correct it and try again. You keep doing this until the program compiles successfully. You now have an *executable* version of your program, i.e., your program has been translated into the internal machine instructions of the computer and the computer can run your program. More specifically, the machine for which the Java compiler generates code is called the *Java Virtual Machine (JVM)*. This machine has not been built in hardware, but instead it is available as software for many operating systems.

Classes in different files must be compiled independently before they can be used. When you write `javac MyClass.java`, the Java compiler will try to compile all the classes that are used directly or indirectly by `MyClass`, but for which there is not yet a `.class` file.

Classpath. We have already seen that we can use classes (like `Point` or `Person`) that are in the same directory as the code where we work with objects of these classes (e.g., by writing something like `Point point = new Point(1,1)` in the `main` method of a class `PointTester` in a different file).

You may now be wondering how it is possible to use classes like `String` that are not in your current directory in the form of a `.class` file. This is because Java comes with a lot of classes built-in, including `String`. These classes are in your CLASSPATH, which is a list of locations in your computer[6] where Java looks for the classes you are using in your program. We will see more about this in the future.

The Java Library

Java is not just a programming language (a set of keywords and syntactic rules). It also provides a broad library of classes that you can use in your program. You know some of them already, like `String` or the boxed types (`Integer`, `Double`, etc).

This library is available for any Java programmer for free. There are literally thousands of useful classes in the library, and they are well documented in one website, colloquially referred to as "the JavaDoc" or "the Java API". You can find it very easily by using your favourite search engine to look up "java API" (Figure 1). API stands for Application Programmer's Interface.

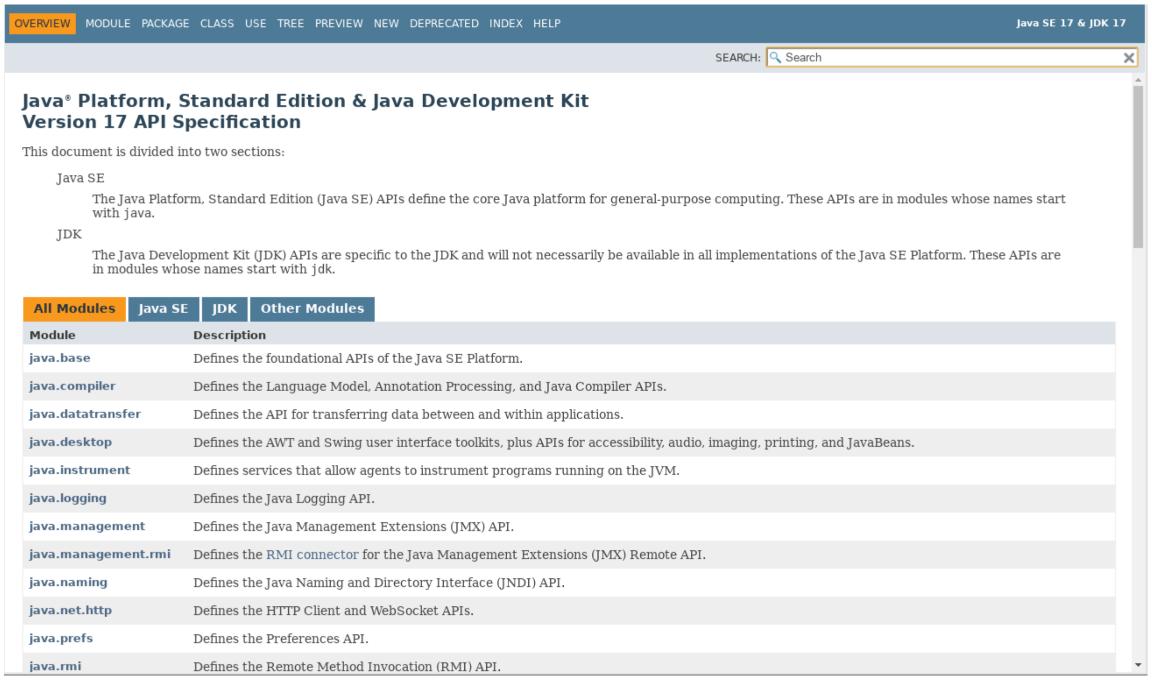
You can find a lot of information about all classes in the Java Library on this website. You can also get a local copy on your hard disk if you plan to work without an internet connection.

Look up any class (e.g., `String`) in the "Search" box on the top-right of the page. Read the documentation. Every page in Java Doc follows the same structure:

General description: at the beginning.

List of fields: all visible fields of the class, if any.

[6] A CLASSPATH is something conceptually similar to a PATH (a list of locations in your computer where the operating system looks for executable files when you type them in the command line). Both are environment variables, and can be accessed and modified in the same way.



Java® Platform, Standard Edition & Java Development Kit Version 17 API Specification

This document is divided into two sections:

Java SE
: The Java Platform, Standard Edition (Java SE) APIs define the core Java platform for general-purpose computing. These APIs are in modules whose names start with `java`.

JDK
: The Java Development Kit (JDK) APIs are specific to the JDK and will not necessarily be available in all implementations of the Java SE Platform. These APIs are in modules whose names start with `jdk`.

Module	Description
java.base	Defines the foundational APIs of the Java SE Platform.
java.compiler	Defines the Language Model, Annotation Processing, and Java Compiler APIs.
java.datatransfer	Defines the API for transferring data between and within applications.
java.desktop	Defines the AWT and Swing user interface toolkits, plus APIs for accessibility, audio, imaging, printing, and JavaBeans.
java.instrument	Defines services that allow agents to instrument programs running on the JVM.
java.logging	Defines the Java Logging API.
java.management	Defines the Java Management Extensions (JMX) API.
java.management.rmi	Defines the RMI connector for the Java Management Extensions (JMX) Remote API.
java.naming	Defines the Java Naming and Directory Interface (JNDI) API.
java.net.http	Defines the HTTP Client and WebSocket APIs.
java.prefs	Defines the Preferences API.
java.rmi	Defines the Remote Method Invocation (RMI) API.

Figure 1: The Java API main page at `https://docs.oracle.com/en/java/javase/17/docs/api/`

List of constructors: each of them with a brief description, a list of parameters and other useful information, and a link to a more detailed description.

List of methods: each of them with a brief description, a list of parameters and other useful information, and a link to a more detailed description.

Detailed description of methods: additional details, use cases, etc.

You must become familiar with the Java Doc. You will use it very often as a Java programmer.

Programmers normally learn about how to use someone else's classes by looking at the Java Doc for the classes rather than looking at the source code of the classes (which may not even be available to them, in case of "proprietary" software).

4 A new complex type: Arrays

A String can be seen as a series of characters one after the other. This simple idea can be extended to series of other simple types: not just chars, but also integers and doubles. In Java, this is called an *array*.

Arrays are a way of having several elements of data of the same type; for example, a company could have a payroll program that used an array with the IDs of all its employees (an array of integers). Arrays can be of simple but also complex types; therefore, the aforementioned program could also have an array of Strings to store the names of the employees. Regardless of the type of the array elements, Java's array types are always complex types. This means that the data contained in arrays is stored in the "heap" part of the computer's memory.

An array is declared using *square* brackets. Square brackets are also used to access the elements in the array. Let's see an example using an array of Strings:

```
        String[] employeeArray;
        employeeArray = new String[5];
        employeeArray[0] = "Alice";
        employeeArray[1] = "Bob";
        employeeArray[2] = "Charlie";
        employeeArray[3] = "Dave";
        employeeArray[4] = "Eve";
        System.out.println("Our first employee is " + employeeArray[0]);
        System.out.println("Our company has " + employeeArray.length + " employees");
```

As you can see, sometimes you need a number inside the brackets and sometimes you do not:

- You do not need a number to declare the array, i.e., to tell the computer that you want to have an array (of Strings, in this case). As all declarations, this reserves a box of type "array of Strings", written "`String[]`", and name `employeeArray`.
- In the next line, we reserve a portion of memory to store the actual boxes of type String, and of course we have to specify the length; otherwise **`new`** will not know how much memory to allocate. Note that arrays are an exception and do *not* have a normal constructor, even if they are created with **`new`** (in other words, there are no round parentheses, but square brackets; this happens in Java only with arrays).
- Finally, when we want to access an element in an array, either for reading or for writing/replacing, we also have to specify *which* element we want to access.

All arrays in Java have an integer field called `length` that is equal to the size of the array. This value never changes. When an array is created (using **`new`**), its size is determined once and for all. Of course, we can still assign a different array to the same variable on the stack:

```
1        int[] myArray;
2        myArray = new int[5];
3        myArray = new int[10];
```

In line 2, we create a new array of length 5 on the heap and store a reference to it in the variable `myArray`. In line 3, we create another new array, this time of length 10, and replace the reference in `myArray` by a reference to our new array of length 10. The previous array of length 5 is now no longer accessible.

At midnight, it is the zeroth hour...

It is very important to remember that the first element of an array is at position `0`. The last element, accordingly, is at position `length - 1`.

You may remember that the first character of a String was also at position zero. As a matter of fact, Strings have classically been implemented internally as arrays of **char**.

If you try to access an element of an array (or a String) that is at position length or beyond, or at a negative position, you will get a runtime error, in the form of an ArrayIndexOutOfBoundsException. A common error is trying to access the last element in the array using myArray[myArray.length] instead of myArray[myArray.length - 1].

Since we can use the integer field length to say exactly how many elements an array has, a typical way of visiting all elements of an array is to use a **for** loop. For example, we can calculate the sum of all elements of an array of type **int**[] with a method like the following:

```java
    public static int sum(int[] numbers) {
        int result = 0; // the running total of the values seen so far
        for (int i = 0; i < numbers.length; i++) {
             sum += numbers[i]; // recall: writing  x += y;  adds y to x
        }
        return result;
    }
```

Curly-brace initialisation of arrays

Initialising an array element-by-element is boring and takes up a lot of space. There is a special notation using curly braces that allows programmers to initialise an array in one line:

```java
        int[] employeeIdArray = {123, 55, 14, 642, 243};
```

If you write something like this, Java will automatically calculate the length of the array for you and will allocate memory for it and point to it.

2-D, 3-D, and beyond...

You can also have arrays of arrays, in which case they are usually called matrices: 2-D matrices, 3-D matrices, etc. Let's see a 2-D example:

```java
        int[][] matrix;
        matrix = new int[3][3];
        matrix[0][0] = 1;
        matrix[0][1] = 2;
        matrix[0][2] = 3;
        matrix[1][0] = 4;
        matrix[1][1] = 5;
        matrix[1][2] = 6;
        matrix[2][0] = 7;
        matrix[2][1] = 8;
        matrix[2][2] = 9;
```

As you can see, a multi-dimensional array is just an array of arrays. It is initialised in the same way as any other array, and its elements are accessed in the same way as with a 1-D array, only using more indexes. You can also use curly-brace notation to initialise an array in a more compact way:

```java
        int[][] matrix3 =  {{1,2,3},{4,5,6},{7,8,9}};
```

And it is not uncommon to write this in different lines to improve clarity. Remember that in Java semicolons are compulsory, so statements do not really finish at the end of the line; they continue from line to line until a semicolon is reached.

```java
        int[][] matrix3 =  { { 1, 2, 3 },
                             { 4, 5, 6 },
                             { 7, 8, 9 } };
```

4.1 Exercise

Write a small program that asks for the names and IDs of all employees in a small company, and store them in an array of integers and an array of Strings. The company has 10 employees.

Use a loop to go through both arrays and print the names and IDs of those employees whose ID is less than 1000. Use another loop to print the names and IDs of those employees whose name starts with "J" or "S".

5 Static data

Let us talk about the keyword `static`. We have used it already for methods that we wanted to run without creating any objects on which to call them. More generally, this keyword means that the field or method that comes after it does not belong to an object of a class but to the class itself: it is a class member rather than an object member.[7]

Why would we be interested in using `static`?

Static fields There is little use for static fields. There are few cases in which some information of our objects should not be stored in the objects themselves but in the class, and it is common that this information is constant. For example, we could store the species name of an animal: this information does not change from animal to animal:

[7] An attempt at an explanation for the choice of word `static` as a modifier for fields that are class members rather than object members: A static field has exactly one copy in the computer's memory. A field that is not static (i.e., an instance variable) has as many different copies in the computer's memory as there are objects of the class with that field. So similar to the explanation for the use of `static` for methods, for fields this keyword means that the compiler knows a bit more about these fields than about their counterparts without that keyword: in this case, how many copies of this variable will be in the computer's memory.

```java
public class Human {
    private static String speciesName = "Homo sapiens";
    private String name;
    private int age;
    // ...
}
```

Every human has a different name and a different age, but the same species. Making the species name **static** makes it clear to other programmers with little knowledge of biology that the species is the same for all humans.

It also saves a little memory because the variable is stored only once (in the class) rather than many times (once per object). For every name, there is one and only one static variable per class per machine; in other words, there is only one `speciesName` in class `Human` in the whole computer.[8]

You can use a static variable to keep a counter of how many objects of a class have been created, as in the example:

```java
public class Spy {
    private static int activeSpyCount = 0;

    public Spy(...) {
        activeSpyCount++;
        // ...
    }

    public static int getNumberOfActiveSpies() {
        return activeSpyCount;
    }
    // ...
}
```

Every time the constructor of `Spy` is called (using **new**) the counter `activeSpyCount` is increased; we could also think of a `retire()` method that reduces the counter. In any case, it is rarely necessary to keep a count of objects of a class (we can accomplish a similar effect using dedicated *factory objects*[9]), and therefore static variables are seldom used except as constants.

Static methods Static methods are class methods. They do not need to be called from an object,[10] but can be called directly via the class name.[11] You have already seen some examples, such as `Math.random()` and the widely used `Integer.parseInt(String)`.

[8]To be precise, there is only one per *JVM*. There may be more than one JVM in a physical computer, although this is not relevant in most cases and the "there is one static X per machine" is mostly true.

[9]You will learn more on the factory pattern and other design patterns in the *Software Design and Programming* module on the MSc Computer Science.

[10]This is why we have also used static methods as "standalone" methods.

[11]If we call a static method from inside the same class, we can skip the class name and the dot and start with the method name.

Static methods can access only **static** fields if they do not have an explicit reference to an object of the class. As they are run from the *class*, they can access only data that is *in the class* and not *in the objects*. You can think of it in the following way: the class does not know where the objects are stored in the memory, so all instance data (i.e., data in the objects) cannot be accessed. This is why the compiler will complain if you write **this**. inside a static method: it is not clear *which* object (if any) you are referring to. Class data and class methods, however, can be accessed from anywhere, just using the name of the class (as in `Integer.parseInt(String)`).

Rules of thumb It is important to understand what **static** means in Java, because you will find this keyword quite often in your life as a Java programmer. On the other hand, it is a keyword that you will not use that much; and when you do, it will be almost always in one of four situations as explained below.

In other words, if you had to remember just four things about the keyword **static**, remember these four rules:

- Use **static** *sparingly*. If you use it often, you are probably doing it wrong. Thing again about your program, and look for ways to get rid of your static state (i.e., fields/variables).
- If you have static fields, they should be *constants* and they should be **public**. One such example is `Math.PI`: its value is always the same 3.14159...
- If you have static methods, they should be *pure functions*. This means they should not have side effects, and they should return a value that depends only on their arguments. One such example is `Integer.parseInt(String)`.
- If you have a `main` method, it must be static as there is only one per class (at most).

6 The `main` method

We can now finally explain all the ingredients of the "scaffold" in which you wrote your programs in the first days:

```
1 public class Birkbeck {
2     public static void main(String[] args) {
3         // ...
4     }
5 }
```

Line 1 defines a **class**. It is **public** so that it is visible also from outside the file. The name `Birkbeck` is a name that I have chosen for the example – what matters here are the requirement that the public class must be in a file whose name is that of the class, followed by "`.java`" (here: "`Birkbeck.java`"), and the convention that in Java a class name should start with a capital letter.

The `main` method in line 2 is the entry point for running our Java programs. Its header (everything before the first "{" in line 2) is a bit long, but we have already learnt what each of its elements means:

- It must be **public**, because otherwise it cannot be accessed from outside the class (and how would the program launch then?).

- It must be `static`, because it pertains to the class, not to each of its instances (i.e., objects).
- It returns `void`. This is a method that is (usually) never called by the programmer (rather than directly by the JVM), so a return type is unnecessary.
- The only parameter is a 1-D array of Strings, called args (short for arguments). Actually the name is not important. The elements in the array are the parameters passed in the command line, if any. For example, if you run a program with a line like

  ```
  > java MyClass www.google.com 80 true
  ```

  the first element of args (i.e., `args[0]`) will be `"www.google.com"`, the second element (i.e., `args[1]`) will be `"80"`, and the third one (i.e., `args[2]`) will be `"true"`. All of them are Strings, so if you want to use these parameters with a different type, you need to parse them.

Liberating your main method from static restrictions

As code in static methods does not understand anything about instance (i.e., non-static) methods or data, you will not be able to use your instance data in your main method. This seems restricting at first, as in the following example: imagine that we want to start a program from a `BitTorrentDownloader` class, and we want to provide the name of the host and the port on the command line. A first attempt might look like this:

```java
public class BitTorrentDownloader {
    private String host;
    private int port;

    public BitTorrentDownloader(String host, int port) {
        this.host = host;
        this.port = port;
    }

    public static void main(String[] args) {
        host = args[0];
        port = Integer.parseInt(args[1]);
        // with the fields initialised,
        // launching code comes here...
        // ...
    }
}
```

When you try to compile this program, `javac` will complain with the following error:

```
non-static variables cannot be referenced from a static context
host = args[0];
^
port = Integer.parseInt(args[1]);
^
```

One (bad) solution is to make host and port static, but that means that there can only be one of each. This is usually a bad idea in the long run. Maybe in a future version you would like several BitTorrentDownloader objects to work in parallel, or maybe you would like to launch many at the same time and put them in a queue. All of this would be impossible if the fields were static.

Remember: **static** means "only one per class", and programmers usually like to have as many as possible of everything, just in case. As a rule of thumb, **never make anything "static" unless it is absolutely necessary**.

There is another (much better) solution: to instantiate the class inside its own main method, and then to run all the code from a non-static launching method. This method is commonly called run() or launch(). See the example:

```java
public class BitTorrentDownloader {
    private String host;
    private int port;

    public BitTorrentDownloader(String host, int port) {
        this.host = host;
        this.port = port;
    }

    public static void main(String[] args) {
        // 1. Parameter analysis / processing
        String host = args[0];
        int port = Integer.parseInt(args[1]);
        // 2. Object construction / building of main data structures
        BitTorrentDownloader downloader = new BitTorrentDownloader(host, port);
        // 3. Execution of the program
        downloader.launch();
    }

    private void launch() {
        // with the fields initialised,
        // launching code comes here...
        // ...
    }
}
```

As you can see, the main method is very simple and has only three parts: parameter analysis, creation of the main structures, and starting the execution of the program. It is a good idea to follow this structure in every main method. This will keep your main methods simple and to the point. If you notice that your main methods are getting longer than a few lines, make a pause, take a deep breath, and think things through again.

7 Casting: conversion of simple types

Java is a statically-typed language. Once a variable is created, Java will not change its type. If you try to assign it a value of a different type it will complain.

However, there are times in which you have an int but want to transform it into a double to perform floating-point division, or some other similar situation. This what *casting* is for. When you cast an expression, you change its type, as in the following example:

```java
        double d = 1.0;
        int i = (int) d; // puts 1 into i (not 1.0)
```

This code will take the value of d, which is 1.0, and transform it into an int (1) before assigning it to the variable i.

Note that not all castings can be done transparently. If you cast a double into an int, for example, you will lose the decimal part because there is no decimal part in an int:

```java
        double d = 1.8;
        int i = (int) d; // puts 1 into i (no rounding by the cast!)
```

If a conversion is between types where it is never the case that information would get lost from one type to the other, Java does the conversion automatically for the programmer. A common example is writing an int value to a variable of type long or double:

```java
        int i = 1;
        double d = i; // puts 1.0 into d (not 1)
```

In some cases like explicitly doing floating-point division, you may also want to use an explicit cast of an int to a double:

```java
        int i = 3;
        int j = 4;
        System.out.println(i + "/" + j + " is about " + (i/(double)j));
```

Here we have cast only one of the two variables explicitly. When the Java compiler sees that we are trying to divide an int by a double, it will use floating-point division for the and insert code to convert the int value to a double value on its own.

Casting for complex types? There is also a version of casting for complex types, but it is more complicated and is related to type hierarchies. We will see how it works when the time is due.