Chapter 01: Building blocks

This chapter covers some basic concepts of object orientation (class members, what are instances, etc). It also covers some basic types and basic classes for data representation (dates, texts, etc).

Let's go then:

- Class members can be either attributes or methods;
- You define classes, methods and attributes by using Java keywords;
- Keywords inform the compile about scpecific characteristics you want something to have. For instance, a `public class Animal {}` code says to the compiler that `Animal` is a `class`, and have all it's characterists, and is also `public`, therefore being accessible by other classes;
- Attributes must be defined at least by a type and a name;
- A method's signature is given by it's name and parameters types. In the book, the given example is:

  ```
  public int numberVisitors(int month) {
    return 10;
  }
  ```

In which the method's signature would be `numberVisitors(int);` * A *top-level type* is a data structure that can be defined independently, inside it's own file. Classes are top-level :sunglasses:; * Only one type can be declared as public in a same file; * A public types' name **must** match it's file's name;

The *main()* method

> ***NOTE:*** It's fun how these years of complex frameworks that handle complex things for you and spare you from a lot of boilerplating, you can almost forget about this little guy and how important it is.

- Finally the *"Hello World"* code, we're 74 pages deep in the PDF and it's finally here. (Yes I implemented the example, cause I'm such a great student :books:);
- If there's any syntax error, the compiler will let you know;
- Now here's a list of rules that must be followed when trying to declare and run a class:
  - Only one **public** type per file;
  - The public type's name must match the file's name;
  - If the class is an entrypoint to the program, then it must contain a valid *main()* method (using the signature `main(String[])`);
- Dissecating the *main()* method:
  - `public`: Access modifier, tells who can call this method (`public` means any class in the program);
  - `static`: This modifier binds the following member to the class, making so that that member can be accessed by using the class name, instead of a class instance;

- `void`: The return type. Since the *main()* method doesn't actually returns anything, then it's return type is void. The book states that is a good practice to use `void` as return type for methods that intend to change some object's state, and since this is literally the purpose of the main method (change the program's state from started to finished), then is logic that you're going to use it as the return type here;

  NOTE: I got a little curous about the declaration made by the authors above, because I have never really though about it that way. So I played a little with the main method by myself in the example's directory. All the classes I wrote are there, with comments saying either they worked or not. Feel free to try.

- The book goes through the different syntaxes accepted for declaring a String array as parameter type (`String[] args`, `String args[]` or `String... args`);
- There's also an example in how to use such parameters. It's implemented as *ZooParams.java* in the examples directory.

Packages: declaring and using

- Packages essentially help us to organize and share code. It's easier to distribute multiple classes in a single package and it's easier to understand my code if I know from which package each class comes.
- Missing `import` statements may cause a compilation error like: `error: cannot find symbol`;
- Example importing and using the `Random` class is implemented in `NumberPicker.java` file;
- The book states clearly that the exam doesn't try to fool you with invalid { import` statements, but just by reading that I guess I'll stay alert to the imports as well whenever they show up;
- The import wildcard ***** can be used in order to import all the classes in a package. It's important to keep in mind that child packages are not imported;
- Now as we all have learned in college, importing a bunch of unused classes (let's say by using the import wildcard) doesn't slows down your program. It's just usually considered a bad practice, since you may then stumble upon classes amidst the program that you're not familiar with and have a hard time figuring out from which package they're from.
- The `java.lang` package is always imported;
- Classes that are int the same package also don't need to import eachother, they're already mutually accessible;
- Another advantage of Java's packaging system is to avoid naming conflicts between classes from different sources. Book brings the classical example of the `Date` class, which is provided by the `java.util` package and the `java.sql` package both;
- When an used class name is found in multiple imported packages, the compiler will complain. There's an implementation of this scenario in the examples folder, called `Conflict.java`
- Class imports take precedence over wildcard imports. This can be used to avoid ambiguosities like the one in the previous example;
- In the specific scenario in which you need to use two or more classes with the same name in a program, the only way is to use *fully qualified class names*. Implemented in

`ConflictResolved.java`;

- Now, all the code we've written to this point was in the *default* package. This is ok for examples and is wildly used in the exam, but is also a bad practice;
- First packaging example:
  - It's implemented under examples/package1 folder. Contains two folders with one class each, and the classes reference each other. Now running this is not that simple as compiling ClassA then compiling ClassB. You shoult try it though, and see how it behaves;
  - To compile those ones you need to pass both files as parameters to the compiler at the same time: `javac packagea/ClassA.java packageb/ClassB.java`;
  - After that, just run: `java packageb.ClassB`;
- It's always better to keep your bytecode away from your source code. To do so, use the `-d` option of the `javac` command to specify a directory to store your bytecode. If you do so, remember to use the `-cp` or `--classpath` option of the `java` command to specify said directory as the classpath that the JVM will use to search for classes;
- Another good way to compile a lot of classes together is to generate a **JAR file**. A JAR is like a ZIP or a tarball, but for bytecode;
- To create a JAR file, you use the `jar` command. The flags and options are incredibly simmilar to those used by the `tar` command (real Linux bros are chilling right now :penguin:): `jar -cfv myJarFile.jar -C classes/directory .`;

  NOTE: Need to get some sleep, stopped at page 22 of the book (page 87 of the PDF);