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 .`;
- Package declaration is not mandatory (again, if you don't declare a package, the class will belong to the `default` package), but if it is present, it must be the first line of valid code in the file;

What about the objects?

Constructors

- An object is an instance of a class (PROFESSOR, Your First OOP);
- Constructors are special block of code that is called in order to create an instance of a class.

  NOTE: The book doesnt clearly states whether a construct is a method or not, at least not up to this point. I found many different points of view in this matter, but I guess we could call it a *block of* *code* that is executed right after field initialization.

- I skipped the first example because it was way too simple, just calling a constructor and storing the reference to the object onto a variable;
- Constructor implementation example in the class `Chicken.java`;
- The test may try to fool you by capitalizing the first letter of a method, like `public void Chicken()`. Beware not to take that for a constructor;
- A dumb constructor is frequently provided by the compiler, if you don't declare any;

Member fields

- Not much to see here, it just states that fields can be read and modified;
- Encapsulation is not covered until much later on the book, so I guess theres really not much to say about fields yet;
- One interesting thing though is that fields values can be read right after they are initialized. See a funny example of this in `Name.java` file;

Initializer blocks

- Blocks are marked by bracers;
- Intance initializers are code blocks defined inside of a class, but outside of a method;
- You can define code blocks inside of a method. In this case, the block will run once the method is called, if it's reached;
- There's an example of different types of code blocks in `Bird.java`;
- The initialization order of a class is always:
    1. Instance initializers and fields, in the order they appear in the code;
    2. Constructor;

Data Types

- There are only two types of data in Java: primitives and references;
- Primitive types are single values in the memory. They are not objects, they are much simpler than that;
- The eight primitives in Java are:
    - `boolean`;
    - `byte`;
    - `short`;
    - `int`;
    - `long`;
    - `float`;
    - `double`;
    - `char`;
- There's no need to memorize the sizes of each type, since this won't be on the test;

- The book covers the very basics on this data types, like how to declare `long` literals by placing a `L` or a `l` at the end of a number;
- There's also an example about the underscores (like `int million = 1_000_000;`). Remember you can't use underscores at the beginning or at the end of a number, nor by a decimal;
- Using numbers in different bases:
  - Decimal: default, no need to specify anything;
  - Binary: use the `0b` prefix; Example: `int numberFive = 0b101`;
  - Octal: use the `0`prefix. Example: `int numberFifteen = 017`;
  - Hexadecimal: use the `0x` prefix. Example: `int twoHundredAndFiftyFive = 0xFF`;
- Now reference types are much more interesting;
- We can think about reference types as Java's weird little *"pointers"*. Just remember that in Java you'll never know the actual memory address that this so-called pointer is storing;
- Reference types alwais point to a memory address containing an object, not a primitive. Said object may store lots of primitives in it's member fields, but it can't be a primitive itself, since it's an object;
- Reference type variables can be assigned `null`, meaning that they're not referring to anything. Primitive ones can't do that;

Wrapper classes

- Every primitive has a corresponding **wrapper class**. Think about wrapper classes as the object version of a primitive, or as a like to call them, primitives on steroids :muscle:;
- Wrapper classes usually have the same name of it's primitive correspondant, but starting with a capitalized letter (`boolean`'s wrapper is the `Boolean` class, etc); The only exception (:notes: *you are the only exceptioooon* :notes:) is the `char` wrapper, which is called `Character`;
- You can "wrap" a primitive by calling it's wrapper's `valueOf` static method, like in `Boolean.valueOf(primitiveBooleanVariable)`;
- All the wrappers, apart from `Boolean` and `Character`, inherits from the `Number` class. This means that they contain very useful methods to easely convert the wrapped value to another numeric type. Example: `byte byteValue = Double.valueOf(doubleVariable).toByte()`

Strings are somewhat special

- Strings aren't primitives, they're reference types just like any other object;
- You can write multiline Strings by using scape characters or by writing text blocks;
- Text blocks are marked by triple double quotes (`"""`, Java's MVP :basketball:). It's important to keep in mind that you **must** break the line right after openning the block;
- Inside a text block you can write whitout worrying about scaping quotes

or any other character;
- Book takes a few paragraphs explaining the difference between *incidental* and *essential* whitespaces. The first one is a space that appears in the text block just for code readability, and the second is actually a meaningfull whitespace;
- You can use a backslash at the end of a line to say that there shouldn't be a break in there;

Java's Identifiers

- Identifiers are the names of variables, methods, classes, interfaces and packages;
- The rules for identifiers are:
  - Identifiers can begin with a letter, currency symbol or an underscore. Any other character leads to a compilation error;
  - The other characters beyond the first can be numbers;
  - A single underscore can't be an identifier;
  - Identifiers can't be the same of any of Java's reserved words;
- About the last one, book states that the exam will only ask you about the most popular reserved words, so there's no need to memorize all of them;
- The words `const` and `goto` are reserved, but are not used in Java;
- The exam will often avoid the naming conventions (*snake case* for constants, *camel case* for the rest) in order to trick you;

Variables

- To declare a variable you just need to define it's type and name. To initialize a variable, you use the equal sign followed by the value you're asigning;
- You may declare multiple variables in the same line of code if they share the same type, separating them with a comma. Example: `String s1, s2, s3;` declares three String variables;
- You bay declare AND instantiate multiple variables in the same line, following the same principle. Example: `String s1, s2 = "hello", s3;`
- A *local* variable is a variable defined within a constructor, method or initializer block;
- Defining a *final local variable* is the same of declaring a final variable with a bigger scope, but it's only valid inside of the local scope. Book makes a big deal about this, so it's probably important;
- Local variables **must** be initialized before they're used, otherwise the compiler will complain. The exam may try to confuse you with that, so it's better to watch out for variables that are declared but not initilized in the same line;
- Parameters behave as *local* variables, in the sense that they only exist within that method or constructor;
- Variables that are not local must be *instance variables* or *class variables.*

The first ones are nothing but fields, and the second ones are variables which are shared by every instance of that very same class;
- It's nice to remember that both of the variables above do not require initialization, because they always have a default value;
- No here comes the `var` part. It's used instead of a variable's type in it's declaration, in order to infer the type;
- There are a few rules:
  - The formal name of this feature is *local variable type inference*, which means that it only works in *local* variables;
  - After the variable is first initialized, it's type is inferred and can't be changed. This is not JavaScript, we're still civilized;
  - The variable must be initialized in the same line of code it's being declared. Doing otherwise will lead to a compilation error;
  - They can't be used in parameters, since they aren't actually *local* variables;
- `var` isn't, surprisingly, a reserved word. You can't use it as a type identifier (you can't create a `class var`) though, because it's a *reserved type name*;

Variable scopes

- This subject has a lot of attention from the authors because it seems to be a good way for the exam to trick us;
- Local variables can never have a scope larger than the block in which they are defined in, but they can always have a smaller scope if you want;
- Conditionals, loops and even plain code blocks are ways to limitate a variables scope;
- Questions containing pieces of code with a very complex logic that won't even compile because of an out of scope variable are pretty common;
- A good way of tracing the scopes is taking notes on the numbers of the lines in which they're openned and closed. This way, when you look to a variable, you can easely assign it to the correct scope;
- This is a very good skill during the test, so practice it!
- With *instance* and *class* variables, everything is much easier. They are accessible right after being initialized, by the whole class;
- Quick list of scopes:
  - *Local variables*;
  - *Method parameters*;
  - *Instance variables*;
  - *Class variables*;

The Garbage Collector

- All Java objects are stored in the *heap* memory space, also refered to as *free store*;
- Once the heap size is all taken, your Java application is out of memory;
- *Garbage collection* is the proccess of freeing memory on the heap space by

destroing unused objects;

- The exam won't ask you about the different garbage colletion algorithms provided by the JVM;
- The only think you do need to know is how to identify if an object is elegible for being destroyed by the garbage collector;
- The method `System.gc();` is a way to suggest to the JVM that a gargabage colletor should run. The JVM is free to ignore it though;
- There are two conditions that make an object *unreachable*, thus elegible for collection:
  - There's no reference pointing to that object anymore;
  - All the existing references to that object are out of scope;
- So the key here is in the difference between an object and it's reference. Garbage collection is all about the references;
- A good idea during the exam is to draw that old fashioned box system that we use in college when first learning C's pointers;