**General Java Basics**

* *Compiler Behavior* 
  + To Compile a java class: Zoo.java **javac zoo.java,** when you run this command, a Zoo.class file gets generated at the location of the Zoo.java file.
  + To run the compiled Zoo.class :   
    **java zoo**
  + You need a JDK to compile because it includes a compiler.
  + You don’t need a JDK to run the compiled class, just the JRE is enough.
  + You can list more than one filename in the javac command. The following command compiles three files:
  + **javac TestProgram1.java TestProgram2.java TestProgram3.java**
  + You can use a wildcard to compile all the files in a folder, like this:

**javac \*.java**

If you need to compile a lot of files at the same time but don’t want to use a wildcard (perhaps you want to compile a large number of files but not all the files in a folder), you can create an argument file

**javac @TestPrograms**

* + You can only have one public class or an interface in a file. **If you have two classes in a java file, the compiler creates a .class for each of those classes**.
  + All command line arguments are treated as strings in the main method
* *Importing Packages*
  + java.lang package is automatically imported.
  + When a class is found in multiple packages, java gives a compiler error: Type is ambiguous  
    Ex: **import** java.util.\*;  
    **import** java.sql.\*;  
    **public class** Basics {  
     **Date** **date**; // !! COMIPLER error because Date is found in both util and sql package.   
    }
  + But this is ok because we have explicitly imported the Date class from util package   
    **import** java.util.Date;  
    **import** java.sql.\*;  
    **public class** Basics {  
     Date **date**;  
    }
  + If two classes are in the same package, you do not need to explicitly use an import statement to access the members of these classes.
  + The import static is used to only import static methods or any static members in a class.
  + Following are not valid declarations for importing static class members
    - **import static java.util.Arrays; // INVALID**
    - **import java.util.Arrays; // VALID because no Static keyword**
  + This would not work:

**import** java.util.Arrays.\*;

{

List<String> strList = Arrays.asList(**"T"**); **// COMPILE ERROR**

OR   
 List<String> strList = asList(**"T"**) **// COMPILE ERROR**

}

Because you would need the import static **java.util.Arrays.\***; or import static **java.util.Arrays.asList**; import for this code to compile.

* + Easy way to remember the difference between a static and non-static import is that if you have an import.static, you can’t just use the class name at the end of import, you have to include at least a .\* or the member name.
  + You can’t just include the package name at the end of an import. You have to either use a.\* or the Class name for non-static imports.  
    ex: import java.time; gives a compile error.
  + A wildcard ending an import statement means that you want to import all the classes in that package. It does not include packages that are inside that one.
* *Variables* 
  + Three types of reference variables in java: Class (Static), Instance (Non-Static) and local (Inside the method).

**Instance Variables (Non-Static Fields):** Technically speaking, objects store their individual states in "non-static fields", that is, fields declared without the static keyword. Non-static fields are also known as *instance variables* because their values are unique to each *instance* of a class (to each object, in other words); the currentSpeed of one bicycle is independent from the currentSpeed of another.

* + - Instance variables have a default value associated to them even without declaration, i.e they do not have to be initialized.
    - In terms of access to the instance variable from an instance block: The instance block cannot pass the instance variables in a method if the instance variables are declared after the instance block. The instance block can assign the instance variable some value, but cannot pass them to a method or use their reference.
    - The constructor and non-static methods can access the instance variables even if the variables are defined after the constructor and methods.
    - Instance variables cannot be accessed by static block or methods
    - A final instance variable can be instantiated in one of three places
      * At the point of declaration like: final int val = 0;
      * In the instance block
      * In the constructor BUT if the final variable is already instantiated in the instance block, the compiler throws an error
      * **If the class has multiple constructors, the final variable needs to be instantiated in each of the constructor or else the compiler throws an error**
* **Class Variables (Static Fields)** A *class variable* is any field declared with the static modifier; this tells the compiler that there is exactly one copy of this variable in existence, regardless of how many times the class has been instantiated. A field defining the number of gears for a particular kind of bicycle could be marked as static since conceptually the same number of gears will apply to all instances. The code static int numGears = 6; would create such a static field. Additionally, the keyword final could be added to indicate that the number of gears will never change.
  + - A static variable can be accessed by non-static methods, constructors and instance blocks but a static block or method cannot access instance variables
    - Order of initialization matters for static instance blocks and static variables, i.e you can’t reference a static variable or pass it to a method if the static variable appears after the static block. You can however assign value to those variables.
    - Constructors, instance methods and static methods can however access static variables even if they are declared afterwards.
    - You can redeclare instance variables in a static block with other values even if they have the same name.
    - A final static variable can only be initialized in one of the two places
      * At the point of declaration
      * At the static block

**A final static variable CANNOT be initialized in the constructor or the instance block**

* + - If the static block where the final variable is initialized comes after the instance block, the instance block cannot use this static final variable.
    - Method and Constructors can use the static final variables regardless of where the final variables are located if they are initialized.
    - Class variables other than a final variable if not initialized have a default value
* **Local Variables** Similar to how an object stores its state in fields, a method will often store its temporary state in *local variables*. The syntax for declaring a local variable is similar to declaring a field (for example, int count = 0;). There is no special keyword designating a variable as local; that determination comes entirely from the location in which the variable is declared — which is between the opening and closing braces of a method. As such, local variables are only visible to the methods in which they are declared; they are not accessible from the rest of the class.
  + - You can redeclare the local variables with the same name as a class or instance variable but they cannot share the same name if there is a parameter with that same name in the method that the local variable is being used.
    - **Local variables must be initialized before they get used**.
    - **Local variables cannot have access modifiers and cannot have the static keyword**
    - Final keyword is allowed for local variable and the they don’t need to be initialized at the point of declaration, they can be done on the subsequent lines.
    - Order of the variables matter, you can’t pass the variables to a method or reference them before initializing them.

After Java 7, you can have underscore in numbers you can add underscore anywhere except beginning, end, right before a decimal point or right after a decimal point. For special primitives such as floats and doubles that require letter in the end, you can’t put an underscore before the word, i.e. you can’t do this

**long l** = 990\_l // COMPILE error

* **You can’t assign null to primitives**   
  **int inty** = **null**; //!! Compile error
* Primitives do not have any methods
* You can’t declare variables like this

**Int I , int J ; // Compiler error**

**int i**,**j**,**k**,**l**,**m**; // VALID , but note none of the values are initialized yet.   
**int n** = 0,**o** = 1,**p** = 8;  
**int q** ; **int r**;  
*// int s , int t; !! COMPILE ERROR*

For wrapper types of primitives of float, double and long we need the letter (F,D,L) at the end of the number

**float** f = 1; OK   
*//Float F = 1; //!!* ***Compile error , found int***

Float F = **new** Float(1.0f);  
Float F2 = **new** Float(1);  
Float F1 = **new** Float (**"1F"**);  
Float F3 = 1.0f;

Notice the println line will throw compile error because l variable is not initialized yet.

**public static void** main(String[] args){  
 **int** i,j,k,l,m;  
 System.***out***.println(l);  
}

Name in Java must begin with letters, Symbols: \_ or $ and subsequent characters can be numbers but they cannot start with numbers.