OCP Java SE 11 Developer Master Cheat Sheet

Introduction

This PDF contains complete notes about OCP Java SE 11 Developer certification.

Notes

Chapter 1: Overview

Candidates who hold this certification have demonstrated proficiency in Java (Standard Edition) software development recognized by a wide range of world-wide industries. They have also exhibited thorough and broad knowledge of the Java programming language, coding practices and utilization of new features incorporated into Java SE 11

By passing the required exam, a certified individual proves tremendous fluency in Java SE and acquisition of the valuable professional skills required to be a Java software developer. This includes a deep understanding of object-orientation, functional programming through lambda expressions and streams, and modularity.

Chapter 2: Java building blocks "you have to learn to walk before you can run"

Constructors

A constructor allow us to create instance object for a given class. There are two rules that a constructor should match : 1. the name of the constructor matches the name of the class. 2. there’s no return type The compiler provide a "default" constructor that do nothing !

Instance initializers

Is a block {} that appears *outside* a method, example :

```
public class Post {
    public static void main(String... args) {
        System.out.println("I'm the main method");
        Post post = new Post();
```

```
    }
    public Post() {
        System.out.println("Hello I'm the constructore");
    }

    {
        System.out.println("Hello I'm an instance block, Before the constructor! I
can call fields like name=" + name);
    }
}
```

Instance initializers are executed before the main method. Any variable that is declared inside a block will not be accessible from outside.

Static blocks

Are blocks {} that are prefixed with static, at class level, theys are executed before anything else on the class. Exemple :

```
public class Test {
    String name = "Abdel";
    public static void main(String... args) {
        System.out.println("I'm executed before anything in the main");
        Test testObj = new Test();
    }

    public Test() {
        System.out.println("Hello I'm the constructore");
    }

    {
        System.out.println("Hello I'm an instance block, Before the constructor! I
can call fields like name=" + name);
    }

    static {
        System.out.println("I'm a static block, I'm executed before instance
initializers");
    }
}
```

Order of initialization

```
1. Static blocks
2. Field and initializer blocks (in the order in which they appear in the file).
3. Constructor
```

Data types

Java support two types of data: * primitive types. * reference types

Primitive types

Java has 8 built-in data-types (the Java primitive types). A primitive is just a single value in memory, such as number or character.

boolean	**true/false**
byte	8 bit => signed (-/+) number [-128,127]
char	16 bit => unsigned number; only positive number are supported
short	16 bit => signed number
int	32 bit
long	64 bit
float	32 bit => require f following the number
double	64 bit

In Java, the data type used to store characters is `char`. A key point to understand is that Java uses Unicode to represent characters. Unicode defines a fully international character set that can represent all of the characters found in all human languages.
When a number is declared, example: `int sum;` java will reserve 32 bits memory. We can use long values with capital 'L' (prefered) or 'l' if the value > int ; otherwise we can ommit the suffix !
you can use *literals* with _ in to assign big values and keep it readable; example :

```
int million = 1_000_000;
```

Reference types

A reference refers to an object, unlike primitives references do not hold the value of the object they refer to in memory. Instead a reference is like a pointer that stores the memory address where the object is located.

Identifiers (variable names) naming rules

an identifier is the name of a variable, it should respect some naming rules, such as :

```
* identifiers must begin with a letter, a $ symbol, or a _ symbol.
* identifiers can include numbers, but should not start with them.
* since java 9, a single underscore _ is not allowed as an identifier.
* you can not use reserved words (java words)
```

By convention java use camelCase notation for naming variables; like `int studentNumber;` you can also use snake_cases for constant (static final) and enum variable names.

Initializing variables

You can declare & initialize variables in the same line, the only condition is that theys should have the same type. Also know that the compiler will not let you use a variable that has not been initialized. Example :

valid declaration

```
String name1, name2, name3 = "Luffy";
```

This will declare 3 variables but only one (name3) is initialized.

Invalid declaration

```
int number = 3, String name = "Luffy";
```

Local variables

A local variable is a variable defined within : constructor, methods, or initializer block. local variables do not have a default value and must be initialized before use.

Instance variables vs class variables

- Instance variables often called field , is a value defined within a specific instance of an object.
- Class variables are defined in class level, and accessible from outside, thus aren't related to an specific instance of the class => they are marked with `static` keyword.

Default values

Variable type	Default init value
boolean	false
byte, short, int, long	0

Variable type	Default init value
char	'\u0000' (NUL)
float, double	0.0
All object references	null

The var keyword (local variable type inference)

Java 10 has introduced the `var` keyword, it replace **local variables** (only local variables) under certain conditions, example :

```
    public void testMethod() {
        var name = "Luffy";
        System.out.println(name);
    }
```

what is not valid with `var`:

```
    public class TestVar {
        var name = "String"; // DOESN'T COMPILE

        public void sayHello() {
            int a = 2, var b = 3; // DOESN'T COMPILE
            var name1 = "Luffy", name2 = "Zoro" // DOESN'T COMPILE => Java does not
allow var in multiple variable declarations.
            var c = null; // DOESN'T COMPILE
            var d;  // DOESN'T COMPILE => type inference is infered on initialization
of the variable
            d = "Hello";
        }
    }
```

The following, is a valid usage of the `var` feature : `var name = (String)null;` since the type is provided !

Rules about type inference

```
* A var is used as a local variable in a constructor, method, or initializer block.
* A var cannot be used in constructor parameters, method parameters, instance
variables, or class variables.
* A var is always initialized on the same line (or statement) where it is declared.
* The value of a var can change, but the type cannot.
* A var cannot be initialized with a null value without a type.
* A var is not permitted in a multiple-variable declaration.
* A var is a reserved type name but not a reserved word, meaning it can be used as an
identifier except as a class, interface, or enum name.
```

Variable scope

There are 3 types of variable scopes:

- Local variables: In scope from declaration to end of block
- Instance variables: In scope from declaration until object eligible for garbage collection
- Class variables (static): In scope from declaration until program ends

Destroying objects

When an object is no more needed it will be destroyed using the garbage collector. The JVM provides multiple kinds of GC. All java objects are stored in the **heap memory**.

Garbage collecting

GC is refered as the process of automatically freeing memory on the heap by deleting objects that are no longer reachable in your program

Eligible for GC

eligible for garbage collection refers to an object's state of no longer being accessible in a program and therefore able to be garbage collected. Java provides a method called `System.gc()` that call the JVM to kick off the GC process, that doesn't means that the GC will perform right away ! The JVM may perform garbage collection at that moment, or it might be busy and choose not to. The JVM is free to ignore the request.

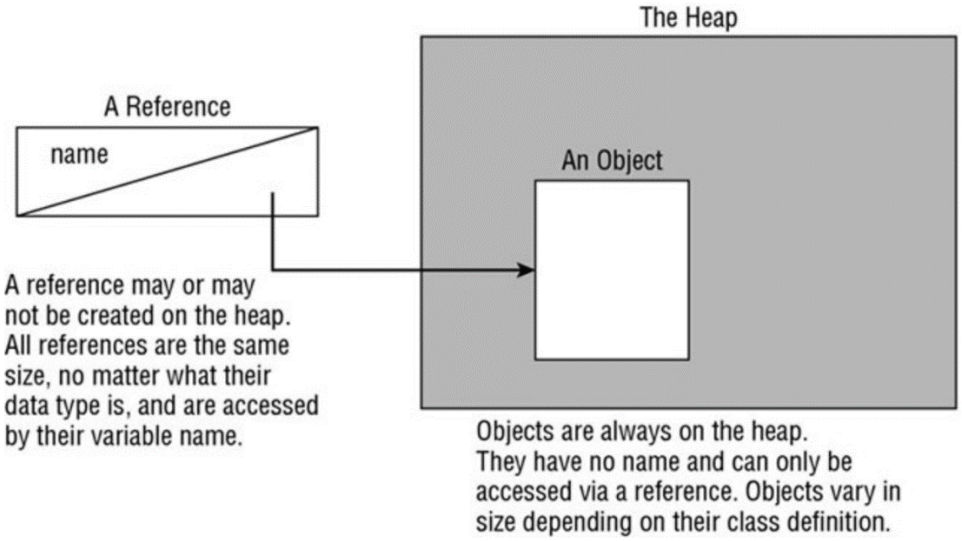
Tracing eligibility

An object will remain on the memory heap until **it's no more reachable/referenced**; when one of two situations occurs:

- The object no longer has a reference point ot it !
- All references to the objects has gone out of scope !

Do not confuse a reference with the object that it refers to; they are two different entities. The reference is a variable that has a name and can be used to access the contents of an object. A reference can be assigned to another reference, passed to a method, or returned from a method. All references are the same size, no matter what their type is.

An object sits on the heap and does not have a name. Therefore, you have no way to access an object except through a reference. Objects come in all different shapes and sizes and consume varying amounts of memory. An object cannot be assigned to another object, and an object cannot be passed to a method or returned from a method. It is the object that gets garbage collected, not its reference.



Before Java 9, java has a finalize() method that was called once by the GC. If the GC didn't run, the finalize method will not be called, if the GC fail to collect the object and tried again later, the finalize method will not be called. But starting from java 9 thid method got deprecated because it causes many issues !

CHAPTER 3 : JAVA OPERATORS

In java, there are 3 flafovers of operators :

- Unary (requires exactly one operand, or variable, or function)
- Binary
- Ternary

Operator precedence

Is when determining which operators are evaluated in what order.

Order of operator precedence

Operator	Symbols and examples
Post unary operator	expression++, expresion--
Pre-unary operator	++expr, --expr
Other unary operator	-,!,~,+, (type)
Multiplication, division, modulus	*, / , %
Addition, substraction	+, -
Shift operators	<<, >>, >>>
Relational operators	<, >, <=, >=, instanceof
Equal to/ not equal to	==, !=
Logical operators	&, ^, pipe
Short-circuit logical operators	&&, 2 pipes
Ternary operators	boolean expr ? expr1 : expr2
Assignment operators	=, +=, -=, *=, /=, %=, &=, ^=, pipe=, <<= , >>=, >>>=

Note that pipe is used instead of | because, md table use pipes to definde columns.

Arithmetic operators

`+ , - , * , / , %`
Those operators are applied to any java primitives, with exepction to boolean. + is also used in String conacatination.
Please note that parentheses override operation order

Floor value For integer values, division (/) results in the floor value (`the value without anything after the decimal point`) of the nearest integer that fulfills the operation, example :

```
double i = 1/2; // 1 ; this is the floor value
double j = 1/2d; // 1.5
```

Number promotion

Rules

1. If 2 values have different data types, java will automatically promote one of the values to the larger of the 2 data types (if possible).
2. If one of the values is integral and the other is floating-point, java will automatically promote the integral value to the floating-point value's data type, but not the inverse.
3. Small data types like byte, char, short are first promoted to int any time they're used with a **java binary aritmetic** operator, so it's preferable to use int rather than byte or short in most cases.
4. After all promotion has occurred and the operands have the same data type, the resulting value will have the same data type as its promoted operands.
5. big literals can be promoted to smaller litterals only if they are defined as `final` and if there values fit in the type of smaller litteral.

example:

```
short a = 10;
short b = 20;
var result = a*b; // result will be an int (the 3rd rule is applied)
```

Casting primitives

casting primitives is required any time you are going from a larger numerical data type to a smaller numerical data type, or converting from a floating-point number to an integral value.

example:

```
int a = 1.2 // 1.2 is a double, so can't compile because double is larger thant int
int b = 1.5f // float > int
short c = 125321154 // out of supported range
```

if we cast those values, it will give us

```
int a = (int)1.2 // 1
long b = (long)19.9f // 19
short c = (short) 125321154 // 16322
```

Another example :

```
short mouse = 10;
short hamster = 3;
short capybara = mouse * hamster;  // DOES NOT COMPILE -> because short values (mouse
and hamster) are automatically prompted to int and the final result is int, which is
> short
```

Please note that even if we cast each value alone `(short)mouse` and `(short)hamster` it will not work neither because casting is an unary oprator, so after casting to short, the binary operator `*` will force the short values to be promoted to int

To fix this example we need to cast the result of `(mouse * hamster)` by adding parenthesis, like his :

```
short mouse = 10;
short hamster = 3;
short capybara = (short)(mouse * hamster);
```

Compound assignment operators

Java support many compound assignment operators such as :

OPERATORS
+=
-=
*=
/=

Compound operators perform a cast of variable before and after applying the corresponding operator, example :

```
int number = 20;
long stock = 2000L;
number *= stock; // number is an int < stock which is a long , so will first cast
number to long then do the multiplication and last cast the result to long
```

Equality operator

In java checking equality of two variables depends on the type of the variables (primitives or objects), one should differentiate between: "2 objects are the same" and "2 objects are equivalents"

| Operator | Apply to primitives | Apply to objects | |------|------| | == | Returns true if the two values represent the same *value* | Returns true if the two values *reference* the same object | | != | Returns true if the two values represent different values | Returns true if the two values do not reference the same object |

when comparing primitives with different types (examples 5 == 5.0) then the smallest data type is promoted to the biggest data type, example

```
char a = 10;
float b = 10.0f;

System.out.println(a == b); // display true, since char is promoted to float and then the comparaison is performed
```

Rules about comparaison : *You can only compare :*

- numeric or character values (but not boolean and numeric or character)
- 2 booleans
- objects, including null and string values (you can even compare a null to a null; null == null)

When comparing objects with ==, the equality is applied to the references to the objects, not to the values of the objects.

comparaison operators:

OPERATORS

>

<

>=

<=

instanceof

the instanceof operator

using when trying to determine the type of a given variable (whether it's a subclass of a given type or implemente a given interface), example :

```
// knowing that Integer inheret from Number we can say

public void printConditionally(Number time) {
    if(time instanceof Integer) {
        System.out.println("I'm an integer");
    }
    // do something here
}
```

The above code can work for any method that have an object parameter that inheret from Number class.

instanceof doesn't work for incompatible types, example : `string instanceof integer` will not compile. This rule is applied to classes but not applied to interfaces (isn't weired huh :p !)
Note also that the expression : `null instanceof Object` returns always false. and that `null instanceof null` doesn't compile.

Logical operators (&,|,^)

Operator	Description
`&`	Logical AND is `true` only if both values are `true`.
`|`	Inclusive OR is `true` if at least one of the values is `true`.
`^`	Exclusive XOR is `true` only if one value is `true` and the other is `false`.

You should familiarize yourself with the truth tables in Figure 3.1, where `x` and `y` are assumed to be `boolean` data types.

x & y (AND)

	y = true	y = false
x = true	true	false
x = false	false	false

x | y (INCLUSIVE OR)

	y = true	y = false
x = true	true	true
x = false	true	false

x ^ y (EXCLUSIVE OR)

	y = true	y = false
x = true	false	true
x = false	true	false

Short circuit operators (||,&&)

Operator	Description
&&	like & (AND) but with a short circuit; if the left side is false, the right side will note be evaluated
||	like | (OR) but with a short circuit; if the left side is true, the the right side will not be evaluated

Common use case (NullPointerException) a common use case for the short circuit operator, is when avoiding NPE (NullPointerException) :

```
Duck duck = null;
if (duck != null & duck.getAge() < 10) {
    //do somthing
}
```

in this case we will get an `NPE` since both sides are evaluated, and we duck is null. To fix this we should replace the `&` (AND) logical operator with `&&` (Short circuit AND), so that when left side (`duck != null`) is false we will not evaluate the right side.
Common unperformed side effects Be wary of short-circuit behavior in some situation, like below :

```
int rabbit = 6;
boolean bunny = (rabbit >= 6) || (++rabbit <= 7);
System.out.println(rabbit); // 6
```

Here since we use `||` short circuit operator, if the left side is true then the right side is skipped. And so `(++rabbit <= 7)` is not evaluated! thus rabbit is always equal to 6.

Ternary operator

is a form of condanced if/else statement, that contains : `(boolean expression) ? expression1 : expression2;`
example:

```
int owl = 5;
int food = owl < 2 ? 3 : 4;
System.out.println(food); // 4
```

CHAPTER 4 : Making Decisions (control flows): if/else, switch statement, while statement, do-while statement, For loop, etc ...

The if/else statement

permit to make a decision in the control flow of execution, **it should contains only boolean expression**, example :

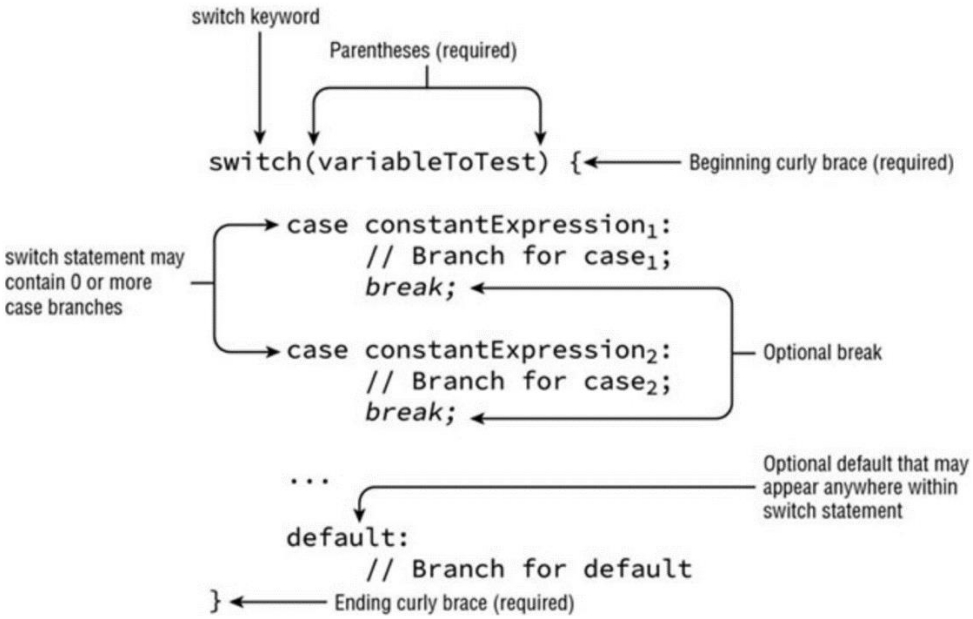
```
int number = 2;
if (number % 2 == 0) {
    System.out.println("even number");
} else {
    System.out.println("odd number");
}
```

The switch statement

The switch statement is used in cases when we have a lot of if statements, for example if we might print a message based on the day of the week:

```
int x = 2;
switch(x) {
    case 2 :
        System.out.println("even number");
        break;
    case 3: case 5:
        System.out.println("odd number");
        break;
    default:
        System.out.println("other");
}
```

structure of a switch statement :



Notice that boolean, long, float, and double are excluded from switch statements, as are their associated Boolean, Long, Float, and Double classes. The reasons are varied, such as boolean having too small a range of values and floating-point numbers having quite a wide range of values. For the exam, though, you just need to know that they are not permitted in switch statements.

allowed types in a switch statment are :

- byte/Byte, short/Short, char/Character, int/Integer, enum (since java 5), String (since java 7), var **if derived from supported type** (since java 10).
- boolean, long, float, double, and their Wrapper classes **ARE NOT ALLOWED IN SWITCH STATEMENT**

switch special cases :

- In a switch statement if no case is mutched and the `default` case is present then the default case will be executed untill it find a `break;` statement, example:

```
/*
    This will display :
    default
    case 3
*/
int x = 2;
```

```
switch(x) {
    case 5:
        System.out.println("case 5");
        break;
    default:
        System.out.println("default");
    case 3 :
        System.out.println("case 3");
        break;
}
```

- The order of the `default` statement is not important. and it's called **only if there is no matching case** !
- Switch cases should be unique (no case duplication is allowed)
- Cases should be **COMPILE TIME CONSTANT**; In case statement, you can use only : literals, enums or `final` constant variables (`final varname = literal`) of **the same type as the variable used in the switch**. Example :

```
final int getCookies() { return 4; }
void feedAnimals() {
  final int bananas = 1;
  int apples = 2; // need final keyword
  int numberOfAnimals = 3; // need final keyword
  final int cookies = getCookies(); // need literal assignement with known value at
compile-time
  switch (numberOfAnimals) {
     case bananas:
     case apples:  // DOES NOT COMPILES
     case getCookies():  // DOES NOT COMPILE
     case cookies :  // DOES NOT COMPILE
     case 3 * 5 :
  }
}
```

- **switch statement support numeric promotion**

```
short size = 4;
final int small = 15;
final int big = 1_000_000;
switch(size) {
   case small:
   case 1+2 :
   case big:  // DOES NOT COMPILE
}
```

The compiler can easily cast small from int to short at compile-time because the value 15 is small enough to fit inside a short. This would not be permitted if small was not a compile-time constant. Likewise, it can convert the expression 1+2 from int to short at compile-time. On the other hand, 1_000_000 is too large to fit inside of short without an explicit cast, so the last case statement does not compile.

- A switch statement is usually more efficient than a set of nested ifs : this is interesting because it gives insight into how the Java compiler works. When it compiles a switch statement, the Java compiler will inspect each of the case constants and create a “jump table” that it will use for selecting the path of execution depending on the value of the expression. Therefore, if you need to select among a large group of values, a switch statement will run much faster than the equivalent logic coded using a sequence of if-elses. The compiler can do this because it knows that the case constants are all the same type and simply must be compared for equality with the switch expression. The compiler has no such knowledge of a long list of if expressions.

- Using switch with string expression is to avoid as you can because it's not so effecient.

WHILE loop statement

A loop is a repetitive control structure that can execute a statement of code multiple times in succession. A while loop until the boolean expression is not met, example :

```
int counter = 10;
while(counter > 0) {
    System.out.println("Hey : " + counter);
    counter--;
}
```

If the condition is not satisfied at the first exection, then while loop body will not be invoked (zero execution).

Note that: The body of the while (or any other of Java’s loops) can be empty. This is because a null statement (one that consists only of a semicolon) is syntactically valid in Java.

```
while(condition); // is a valid java statement
```

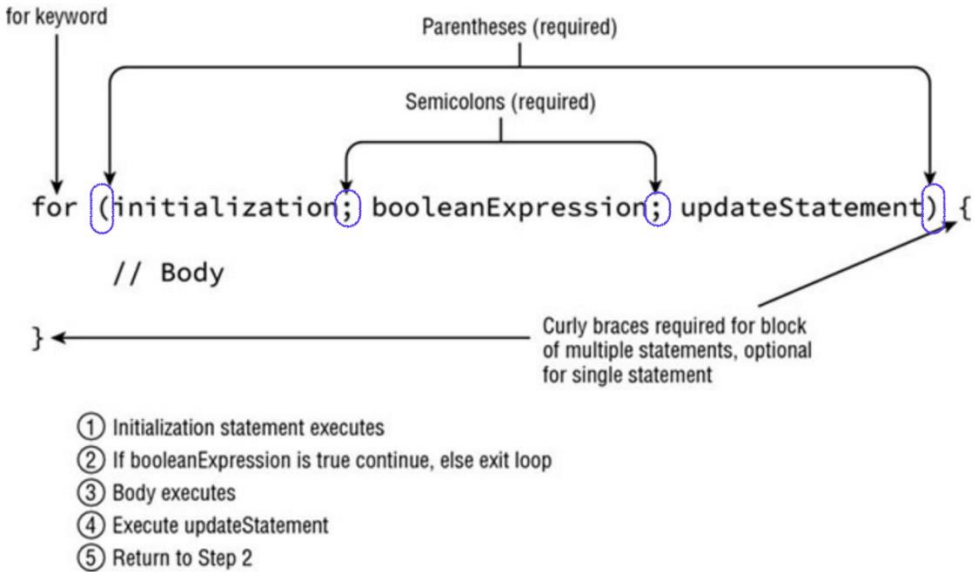
The DO/WHILE statement

Unlike a while loop, though, a do/while loop guarantees that the statement or block will be executed at least once. Whereas a while loop is executed zero or more times, a do/while loop is executed one or more times.

```
int lizard = 0;
do {
   lizard++;
} while(false);
System.out.println(lizard);  // 1
```

The FOR loop

A basic for loop has the same conditional boolean expression and statement, or block of statements, as the while loops, as well as two new sections: an initialization block and an update
statement.



- Variables declared in the initialization block of a for loop have limited scope and are accessible only within the for loop
- we can use `var` (type inference) in initialization block like so :

```
// i will be of int type
for(var i = 0; i < 20; i++) {
    // do something
}
```

- We can add multiple terms (separated with semi colon) to the for loop, like bellow :

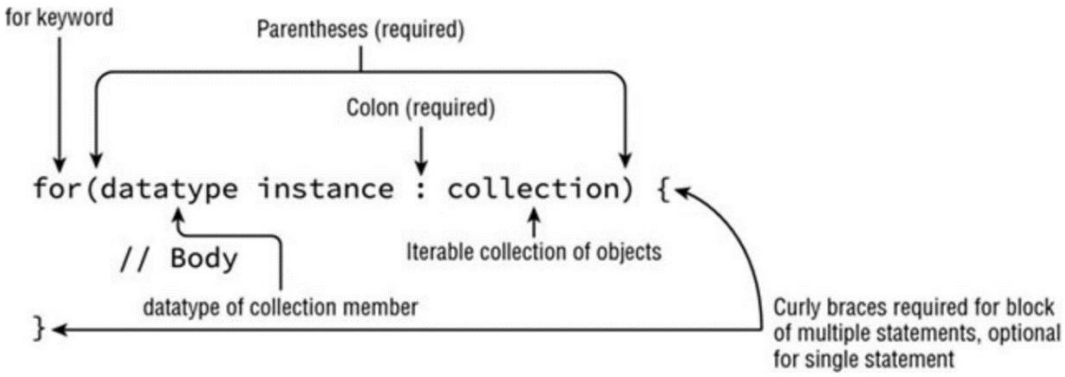
```
int x = 0;
for(long y = 0, z = 4; x < 5 && y < 10; x++, y++) {
   System.out.print(y + " "); }
System.out.print(x + " ");
```

- Variables in the initialization block should have the same type.
- We can't use the same variable name in initialization block if there is already an existing one outside the for loop.

The FOR EACH LOOP

The for-each loop is composed of initalization part (at the left) + the object to be iterated over. The right side must be one of the following:

- A built-in Java array.
- An object whose type implements java.lang.Iterable (this doesn't include all Collection framework classes or interfaces, but only those that implement or extend that Collection interface).



- Java convert the foreach loop into a standard for loop during compilation (for arrays it use indexes, for collection it uses `Iterator` interface).

NETSTED LOOPS

Netsed loops are simply that are inside other loops, example :

```
int[][] twodim = new int[][]{{1,2,3}, {4,5,6}, {7,8,9}};
for(int[] row : twodim) {
  for(int column: row) {
    System.out.print(column + "\t");
  }
  System.out.println("\n");
}
```

OPTIONAL LABEL

A label is an optional pointer to the head of a statement that allows the application flow to jump to it or break from it. It is a single identifier that is proceeded by a colon (:). For example :

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

```
OUTER_LOOP: for(int[] row : twodim) {
  INNER_LOOP: for(int column: row) {
    System.out.print(column + "\t");
  }
  System.out.println("\n");
}
```

The Jump statements (BREAK, CONTINUE, RETURN)

The BREAK Statement

In java a `break` statement has 3 uses:

- It terminates a statement sequence in a switch.
- It can be used to exit a loop (the innermost loop).
- It can be used as a *civilized* form of the goto. The `break` statement transfers the flow of control to the enclosing statement.
- If the break is specified without an optional label, it will terminate the nearest inner loop it is currently in the process of executing !
- If the break is specified with a label, it will terminate a higher outer loop referenced by this label.

```
optionalLabel: while(condition) {
    // body
    for(int a: nums[]) {
        // somewhere in the loop
        break optionalLabel; // this will terminate the while loop
    }
}
```

The CONTINUE statement

Sometimes it is useful to force an early iteration of a loop. That is, you might want to continue running the loop but stop processing the remainder of the code in its body for this particular iteration.

The `continue` statement causes flow to finish the execution of **the current loop**.

- If the `continue` is used with an optional Label, it will stop the execution of the current processing and continue (jump) to the next record in the outer loop where the label is present.
- Otherwise it will stop the execution of the current processing in the nearest inner loop and continue to the next record.

```
optionalLabel: while(condition) {
    // body
    for(int a: nums[]) {
        // somewhere in the loop
        continue optionalLabel; // this will stop the current processing and pass to
the next record in the while condition (if any - depends on the condition)
    }
}
```

The RETURN statement

The return statement is used to explicitly return from a method. That is, it causes program control to transfer back to the caller of the method. As such, it is categorized as a jump statement.

Notes:

In addition to the jump statements discussed here, Java supports one other way that you can change your program's flow of execution: through exception handling. Exception handling provides a structured method by which run-time errors can be trapped and handled by your program. It is supported by the keywords try, catch, throw, throws, and finally. In essence, the exception handling mechanism allows your program to perform a nonlocal branch.

Chapter 5 : Core java API

Creating and manipulating String

A string is basically a sequence of characters (implements the `CharSequence` interface), it's definition look like :

```
public final class String implements java.io.Serializable, Comparable<String>,
CharSequence {
    // ....
}
```

- String is a final class => can't be inhereted
- The String class is a special class that you can initialize in two ways (with and without the keyword`new`) :

```
private String name = "Ali";
private String greeting = new String("Hello");
```

Concatenation

Combining Strings is called concatenation (`"1" + "2" // will result "12"`). There are some rules to know about string concatenation to know when using + operation : 1- if both operands are numeric => addition 2- if one of operands is a String => contatenation
Be aware of this tricky case (+ operation operate from left to right):

```
String result1 = 1 + 2 + "a"; // gives "3a"
String result2 = "a" + 1 + 2; // gives "a12"
```

Immutability

Once a String object is created, it cannot be changed or resized. Mutable word means "changeable", Immutable is the oposite word ! To ensure immutability of a class, it should not provide a way (methods like setters or if its members are public) to changes its state; example :

```
public class MutablePerson {
    private String name;
    public int age; // this make the class mutable, since we can change the object
state through age property

   // The setter make this class mutable since we can change its state by setting a
new name
    public setName(String name) {
        this.name = name;
    }
}

/**
 * This class is immutable since when created through the constructor there is no way
to change its state (member states)
 */
public class ImmutablePerson {
    private String name;
    private int age;

    public ImmutablePerson(String name, int age) {
        this.name = name;
        this.age = age;
    }
    // only getters here
}
```

- **Immutable classes in java are final**, which prevents subclass creation. You would having a subclass adding mutable behavior.

- There is another way to concat strings; through `String#concat() method`, this result is a new immutable String, example :

```
String s1 = "1";
String s2 = "2".concat(s1);
s2.concat("s3"); // the result is thrown away
System.out.println(s2); // gives "21"
```

IMPORTANT STRING METHODS

Method	Explination	Example
`int length()`	return the number of characters is the String.	`"hello".length(); //gives 5`
`char charAt(int index)`	return the character at the given index position	`"hello".charAt(1); // returns 'e' "hi".charAt(2); // throw java.lang.StringIndexOutOfBoundsException`
`int indexOf(int ch)` or `indexOf(int ch, int fromIndex)` or `indexOf(String str)` or `indexOf(String str, int fromIndex)`	Looks at the characters in the string and find the first index that matches the desired value, can accept an individual char or a whole string as input, it can also start from a desired position. **INDEX returns -1 if it can't find a matching result**	`"hello world".indexOf("ld",4); // returns 9`
`String substring(int beginIndex)` or `String substring(int beginIndex, int endIndex)` **[begin,end[**	retruns a part of the original string, starting from an index (included) up to the end or up to an end index (excluded). In case of invalid index, the method throw a `java.lang.StringIndexOutOfBoundsException`	`"hello world".substring(6); // gives world "hi".substring(1,1); //gives empty string`
`String toLowerCase()` and	upperCase or lowerCase a string	`"hello".toUpperCase(); // gives "HELLO"`

Method	Explination	Example
`String toUpperCase()`		
`boolean equals(String)` and `boolean equalsIgnoreCase(String)`	`equals()` Verify if two strings contains the same characters at the sae orders, `equalsIgnoreCase()` do the same besides it convert the character cases if needed (ignoreCase)	`"hello".equals("HEllo"); // false - "hello".equalsIgnoreCase("HEllo") //true`
`boolean startsWith(String prefix)` and `boolean endsWith(String suffix)`	look if a string start with a given prefix or end with a given suffix. **Case sensitive**	`"hello".startsWith("he"); // true - "hello".startsWith("He"); // false`
`String replace(char oldChar, char newChar)` or `String replace(CharSequence target, CharSequence replacement)`	replace() method does a simple search and replace on the string	`"abcabc".replace('a', 'A'); // AbcAbc - "abcabc".replace("abc", "Hello")); // "HelloHello"`
`boolean contains(CharSequence charSeq)`	looks for matches in the String	`"abc".contains("b"); // true - "abc".contains("B")); // false`
`String strip()` `String stripLeading()` `String stripTrailing()` `String trim()`	strip and trim are methods that removes whitespace from begin or/and end of string whitespace consist of space,'\r','\n','\t') **strip was introduced in java11, it support unicode also**, stripLeading removes whitespace at the beginning of the string and leaves them at	`" \t hello world \n".strip(); // gives "hello world"`

Method	Explination	Example
	the end, stripTrailing do the opposite	
`String intern()`	return the value from the string pool if its there, otherwise it adds the value to string pool	`"hello".trim()`
`String[] split(String regex)`	Split the string into array of string by a matching regex	`"hello-world".split("-");` `// gives {"hello","world"}`

StringBuilder

- Is a mutable class that provide the possibility of creating and updating strings.
- We used to have in older version the `StringBuffer` which is the same as `StringBuilder` except **it supprts threads**, but it's much **slower than `StringBuilder`** !
- The `StringBuilder` have 3 constructors :

```
StringBuilder sb = new StringBuilder(); // init empty String
StringBuilder sb = new StringBuilder(10); // length of the string
StringBuilder sb = new StringBuilder("Hello");  // init with a given value
```

- The `StringBuilder` use the `StringBuilder append(CharSequence char)` to concatenate Strings, note that **it returns the reference of the StringBuilder** :

```
StringBuilder s1 = new StringBuilder("hello");
StringBuilder s2 = a.append("-world"); // here we modified the object referenced by a
("abc") by adding "efg" to it, now both references a & b points to the same object
b.append("-from").append("-stringBuilder"); // here we modify the object referenced
by a & b
System.out.println("s1 : " + s1); // prints "hello-world-from-stringBuilder
System.out.println("s2 : " + s2); // prints "hello-world-from-stringBuilder
```

IMPORTANT StringBuilder METHODS

Method	Explination	Example
`charAt()`, `indexOf()`, `length()`, and `substring()`	Same as String functions	None

Method	Explination	Example
`StringBuilder append(String str)` it also accept many datatype	Add the parameter into the stringBuilder instance and return the reference to that object	`s1.append("world"); // gives "helloworld"`
`StringBuilder insert(int offset, String str)`	Insert at the given offset/position a given parameter	`s1.insert(0, "prefix")`
`StringBuilder delete(int startIndex, int endIndex)` **[startIndex,endIndex[** or `StringBuilder deleteCharAt(int index)`	Delete a sequence of characters between start and end indexes	`StringBuilder s1 = new StringBuilder("hello"); s1.delete(1,4).toString(); // gives "ho"`
`StringBuilder replace(int startIndex, int endIndex, String newString)`	Works differently from `String#replace(string,string)`, it replace the char sequence between **[startIndex,endIndex[** with a given char sequence	`s1.replace(1,4, "ELL") //gives "hELLo"`
`StringBuilder reverse()`	It reverse the char sequence of stringBuilder	`s1.reverse(); // gives oLLEh`

Equality

Equals() vs ==

When using == operator to compare 2 instances of String/StringBuilder, meaning that we are comparing references not values of objects, since we are dealing with objects.
Examples:

```
String a = "Hello World";
String b = " Hello World".trim();
String c = a.trim().intern(); // intern() looks at the string pool if it can finds the object there, and return it
```

```
System.out.println(x == z); // false
System.out.println(a == c); // true (since a and c references the same object)
```

The authors of `StringBuilder` didn't implements equals() method; if you call `StringBuilder#equals()` on 2 stringBuilder instances, it will check reference equality. You can call `StringBuilder#toString()` and then compare.
Whenever we call an unplemented `quals()` method on an object, it will compare references not objects (values).

THE STRING POOL

Since strings are used everywhere in java, they use a lot of memory. This can leads to leak in memory for big applications. Fortunalty java had added the functionality to reuse those repeated strings.

- The string pool (intern pool) is a location in the JVM where java stocks all **literals and COMPILE TIME CONSTANT strings**.
- The string pool **contains only literal values**, note that `someObject.toString()` and `someStringValue += "world"` is a string but not a literal string.
- Using concatenation is like calling a method and results a new string.

Common cases:

```
String s1 = "Hello World";
String s2 = "Hello " + "World"; // compile time constant => placed to the string pool
if not yet exists as "Hello World". In our case s2 and s1 shares the same string pool
reference.
String s3 = "  Hello Wolrd".trim(); // computed at runtime - isn't the same at
compile time
String s4 = new String("Hello World"); // not a literal value - tels the JVM to
create a new string object and don't use string pool
String s5 = "Hello ";
s5 += "World";
String s6 = s4.intern(); // looks of this string exist in the string pool

System.out.println(s1 == s2); // true
System.out.println(s1 == s3); // false
System.out.println(s2 == s3); // false
System.out.println(s1 == s4); // false
System.out.println(s1 == s5); // false
System.out.println(s1 == s6); // true
```

JAVA ARRAYS

We said that a string is a sequence of characters, this is true because in java a String is implemented as an **array** for characters (`char[] characters;`). *An array is an area of*

memory on the heap with space for a designated number of elements. A `StringBuilder` is implemented as an array where the array object is replaced with the new bigger array object when it runs out of space to store all the characters.

What is an array ?

An array is an ordered list, of the same type, with fixed size. It can conatain duplicates.

Creating an array

1- One way of creating an array is using the `new` keyword and specifiying its size, example :

```
int[] numbers = new int[20]; // all elements of the array takes the default value of the array type, in our case it will be [0,0,0...,0] 0 tweenty time.
```

2- Another way of creating an array is to specify all elements it should start with, example :

```
int[] numbers = new int[] {20,30,40,50,60}; // this time we created an array of 5 elements.
```

Anonymous arrays:

The statement "new int[]" is considered as redundant since we are specifying the type of the array on left side, java already now the type and the size, as simpification java let you do it like this :

```
int numbers = {20,30,40,50,60}
```

This is called an **anonymous array**.

Valid forms of array declaration:

You can use [] before or after reference name, and adding a space is optional. Are those declaration are valides:

```
int[] numbers;
int [] numbers;
int []numbers;
int numbers[];
int numbers [];
int[] a,b; // is equivalent to int[] a; int b[];
int a[], b; // this is equivalent to int a[]; int b;
```

Creating an array with reference:

You can choose any type to be the type of an array, this include classes :

```
String[] cars = {"Bugati", "Lamberguini", "Ferrari"};
String myCars = cars; // got the same reference

System.out.ptintln(myCars.equals(cars)); // true => since we compare the references
not the array content !
System.out.ptintln(myCars.toString()); //
```

The output `[Ljava.lang.String;@160bc7c0` means: [L (it's an array), `java.lang.String` (is the Type of the array), and *160bc7c0* (is the hashcode, it will change on each execution since it's a reference).
Since java 5, java has provided a method to prints all elements in an array called : `Arrays.toString(cars)`

Arrays and memory allocation:

The array doesn't allocate space for objects values. Instead is allocates space for a reference to where the objects are really stored. An array that is not instanciated point to nothing, which means `null`, example : `String[] cars;`. An array like that `String[] cars = new String[2];` have two elements both currently are `null`, but has the potentiel **to point to** a String object.

Casting an Array:

Like we learned before we can go from an object to a more specific object which inheret from the former one. This is also true about array.

```
String cars[] = {"Bugati", "Ferrari"};
Object[] vehicle = cars; // without casting since Object is superiour to String
String[] anotherCars = (String[]) vehicle; // casting is needed, we go from more
specific object to it's parent
anotherCars[0] = new StringBuilder("Mercedes"); // COMPILE TIME ERROR : can't assigne
StringBuilder to String type.
vehicle[0] = new StringBuilder("Bycle"); // RUNTIME EXCEPTION :
java.lang.ArrayStoreException. even if StringBuilder inheret from Object, this will
not work since vehicle reference an array of Strings.
System.out.println(Arrays.toString(cars));
```

Array useful methods :

Method	Description
`void Arrays.sort(cars)`, `void Arrays.sort(T[] array, Comparator<? extends T> comparator)`	Sort any kind of array
`String Arrays.toString(cars)`	Return elements as string `["a", "b"]`

Method	Description
`int Arrays.binarySearch(cars, "Bugatti")`	Search for element only if array is sorted, will returns index of match if exits, otherwise returns a negative value (which is the reverse index of where this could be fit in the array)
`int compare(cars1, cars2)`	Compare 2 arrays; return (-/+indexOfElementInArray (if both arrays length are equals and first none match of cars1 < cars2 => (- signe, otherwise + signe) / if arrays length are different return -/+sizeOfTheBiggestArray),0 if both are equals) `-number => cars1 < cars2
`int mismatch(Object[] array1, Object[] array2), int mismatch(Object[] array1, Object[] array2, Comparator<? extends T> comparator)`	If the arrays are equal, mismatch() returns -1. Otherwise, it returns the first index where they differ

Rules about compare :

- If both arrays are the same length and have the same values => the return = 0
- If cars1 contains the same elements as cars2 but cars2 has extra element => the return of `Arrays.compare(cars1, cars2)` = +indexOf
- If the first element that differs is smaller (in cars1) than the other (in cars2) => the return of `Arrays.compare(cars1, cars2)` = 1 Smaller element are defined follow these rules : null < numbers < strings (UPPER CASES < LOWER CASES)

Varargs (variable argument)

When passing an array to a method,it could be done in one of the following three ways :

```
public void method(String[] args) {}
public void method(String args[]) {}
public void method(String... args) {} // this is called varargs (variable argument)
```

The third way is called `varargs`, a feature that was introduced in JDK5 and it's similar to the other ways; thus we can treate `args` as a regular array, so we can use `args[0]` and `args.length`.

Multi-dimentional arrays

Arrays are objects, and array elements are objects too, so the element it self could be an array, and so on and so forth...

When you allocate memory for a multidimensional array, you need only specify the memory for the first (leftmost) dimension. You can allocate the remaining dimensions separately.

Declaring multidimentional arrays:

We can declare multidimentional arrays like so :

```
int[][] array = new int[2][2]; // 2D array of 2 rows / 2 columns
int[] array[] = {{1,2,3},{4,5,6}}}; // 2D array of 2 rows, 3 columns
int[] array1[] array2[][]; // 2D and 3D array
```

Asymmetric multidimentional array:

While arrays happens to be rectangle in shape, they doasn't need to be so, we can declare mulitdimentional arrays where each element is an array of different size; this type of arrays is called an asymmetric multidimentional arrays. Example :

```
int[][] asymmetricArray = {{1,2}, {3}, {4,5,6}};
int[][] asymmetricArray = new int[5][]; // column might have different sizes
asymmetricArray[0] = new int[6];
asymmetricArray[1] = {1,2};
```

Traversing a multidimentional array:

There is multiple ways for looping over multidimentional arrays;

- Using **FOR** loop :

```
int[][] multidimArray = {{1,2,3},{4}};
for(int i=0; i < multidimArray.length; i++) {
    for(int j=0; j < multidimArray[i].length; j++) {
        System.out.println(multidimArray[i][j]);
    }
}
```

- Using **FOREACH** loop :

```
int[][] multidimArray = {{1,2,3},{4}};
for (int[] rows : multidimArray) {
  for (int column : rows) {
```

```
    System.out.print(column);
  }
}
```

NOTE Keep in mind that the value of length has nothing to do with the number of elements that are actually in use. It only reflects the number of elements that the array is designed to hold.

ArrayList in Java

Java array have some limitations such as fixed size; in order to use an array you should know its size in advance which is not suitable ! Just like `StringBuilder` (for `String`), an `ArrayList` can change size (capacity) at runtime as needed.

- An `ArrayList` is an order list like Array that allow duplicates, the only difference between `ArrayList` and `Array` is size/capacity which is dynamic in `ArrayList` and fixed in `Array`.
- `ArrayList` implement `List` which is an interface, so know that you can store an `ArrayList` in a `List` reference but not the inverse.

```
List<String> list = new ArrayList<>();
```

Creating an ArrayList

Just like `StringBuilder` an, there are 3 ways to create `ArrayList` :

```
// Pre-Java 5 (before Generic) way of creating arrays
ArrayList list1 = new ArrayList();
ArrayList list2 = new ArrayList(10);
ArrayList list3 = new ArrayList(list2);

// After Java 5 (generic) way
ArrayList<String> list4 = new ArrayList<String>();
ArrayList<String> list5 = new ArrayList<>(10); // Starting from java 7 we can omit
the Type in the right side of declaration / the <> (called also diamond operator) are
still required.
```

Using var with ArrayList

We can use the **local variable type inference** `var` with `ArrayList` as follow :

```
var array1 = new ArrayList<String>(); //
array1.add("hello");
array1.add(1, "world");
// This will print : hello  world
for(var element : array1) {
    System.out.print(element + "\t");
}
```

A declaration like this : `var array1 = new ArrayList<>()` is authorized, the `array1` can accept any object type. but can only be traversed using Object element :

```
var array1 = new ArrayList<>();
array1.add("Hello"); // pass because Object (is the most generic) > String
```

```
array1.add(1L); // since it's an array of object and Long is Object too
for(String element: array1) {} // doesn't COMPILE , we need to use Object to pass it
because, array1 is of ArrayList<Object>, so only object can pass (String < Object)
for(Object element: array1) {} // This pass successfuly
```

Using an ArrayList

`ArrayList` has many methods, among them there are (we will suppose that we already initialized an ArrayList called list; `List<String> list = new ArrayList<>(); list.add("Hello");`):

Method	Description	Example
`boolean add(E element)`, `void add(int index, E element)`	This method add elements to the `ArrayList`. The first signature always return `true` the second one adds the element at the specific index/position and moves existing elements to the right (see the example)	`list.add("World"); list.add(0,"Hi ! "); // gives : Hi ! World Hello`
`boolean remove(Object object)`, `E remove(int index)`	Remove the first matching element (case sensitive) in the ArrayList (return `true` if matched and removed otherwise `false`), or remove the element at a specific index/position, if the index doesn't exist this will throw `java.lang.IndexOutOfBoundsException`	`boolean isRemoved = list.remove("Hello"); String removedElement = list.remove(0);`
`E set(int index, E newElement)`	Change an element at the specific index/position, without changing the ArrayList size, and return the element that got replaced (old one)	`String replacedElement = list.set(0, "Hallo"); // replace Hello by Hallo`
`boolean isEmpty()`	Return a boolean to tell whether a list is empty or not	`boolean isEmpty = list.isEmpty(); // gives false since the list contains "Hello"`
`int size()`	Return the size of the ArrayList	`int size = list.size(); // return 1 since the`

Method	Description	Example
		`list contains one element "Hello"`
`void clear()`	Remove all elements of a list	`list.clear();`
`boolean contains(Object object)`	Check whether a certain value is in the `ArrayList`. this method calls `equals()` on each element of the `ArrayList`, since String implements `equals()` this works out well.	`list.contains("Hello"); // return true`
`boolean equals(Object object)`	ArrayList has a custom implementation of `equals()`, so we can compare 2 lists to see if **they contain the same elements in the same order**, the methods calls `equals` for each element at the same position in both list to compare equality.	`list.equals(new ArrayList<>()); // return false`

Converting between ARRAY and LIST

Convert List to Array

To convert from a `List` to an `Array`, we use the method `Object[] toArray();`, this will return an generic array `Object`, if we want to have a specific type array we should specify the type inside the method`toArray(new Type[size])`, Generally, we specify a size of 0 to the Type array, The advantage of specifying a size of 0 for the parameter is that Java will create a new array of the proper size for the return value. If you like, you can suggest a larger array to be used instead. If the ArrayList fits in that array, it will be returned. Otherwise, a new one will be created:

```
List<String> list = new ArrayList<>();
list.add("hello");
Object[] genricArray = list.toArray(); // this make a copy of the original list
String[] stringArray = list.toArray(new String[0]); // this make a copy of the
original list
list.clear(); // now list.size()==0
System.out.println(genricArray.size()); // print 2
System.out.println(stringArray.size()); // print 2
```

Convert Array to List

There are 2 ways of converting Array to List, each has special behaviour to learn about :

1- Backed list : One option is to create a List that is linked to the original array. When a change is made to one, it is available in the other. It is a **fixed-size list** and is also known as a backed List because the array changes with it:

```
String[] array = { "hello", "world" };     // [hello, world]
List<String> list = Arrays.asList(array); // return fixed size list, size = 2
list.set(1, "WORLD"); // [hello, WORLD]
array[0] = "HELLO"; // [HELLO, WORLD]
System.out.println(Arrays.toString(array)); // [HELLO, WORLD]
list.remove(1); // throws UnsupportedOperationException (since it's a fixed size)
```

2- Immutable list : Another option is to create an immutable List. That means you cannot change the values or the size of the List. You can change the original array, but changes will not be reflected in the immutable List.

```
String[] array = { "hello", "world" };     // [hello, world]
List<String> list = List.of(array); // return immutable list, with size = 2
array[0] = "HELLO"; // [HELLO, world]
System.out.println(list); // [hello, world]
list.set(1, "WORLD"); // throws UnsupportedOperationException (since it's an
immutable list)
```

Using Varargs to Create a List

You can also use varargs to create list in a simple way using the `List.of(Object... o)` or `Arrays.asList(Object... o)`:

```
List<String> cars = Arrays.asList("Mazeratti", "Bugatti"); // Backed list, with fixed
size
List<String> brands = List.of("Nike", "Louis Vuitton"); // Immutable list
// or also
List<String> players = List.of({"Messi", "Ronaldo"});
```

Sorting an ArrayList

This is similar to sorting an `Array` (where we used `Arrays.sort()`). There are 2 methods to sort an ArrayList we can do it :

Using `Collections.sort(List)` or `Collections.sort(List, Comparator)`

The use these methods, the List should be modifiable, but not resizable.

Using `Collections.sort(List)` will sort the list in the natural ascending order (since the comparator is null) and the element of the list should implement `Comparable` interface.

Using `List#sort(Comparator<? super E> c)` method.

The use this method, the List should be modifiable, but not resizable. the implementation simply transform the List to an Array and uses `Arrays.sort(T[] array,`

`Comparator c)` behind the scene. if the comparator. If the `Comparator` in argument is null, this will leads to ordering the list in natural order, thus the element of the list must implements the `Comparable` interface.
Examples :

```
List<String> cars = Arrays.asList("Rolls Royce", "Bugatti");
List<String> brands = Arrays.asList("Nike", "Louis Vuitton");
Collection.sort(cars): // ascending, natural order
brands.sort(null); // ascending, natural order
System.out.println(cars);
```

Creating Sets and Maps

...

WRAPPER CLASSES

In java each primitive type have a wrapper class, which is an object that corresponds to the primitive :

Primitive type	Wrapper class	Example
`boolean`	`Boolean`	`Boolean.valueOf(true)`
`byte`	`Byte`	`Byte.valueOf((byte)1)`
`short`	`Short`	`Short.valueOf((short)1)`
`char`	`Character`	`Character.valueOf('c')`
`int`	`Integer`	`Integer.valueOf(1)`
`long`	`Long`	`Long.valueOf(1)`
`float`	`Float`	`Float.valueOf((float)1.0)`
`double`	`Double`	`Double.valueOf(1.0)`

Wrapper classes methods

`Wrapper classes` are immutables that encapsulate a the value of a primitive type, and add OO feature to it.

The valueOf method

Wrapper classes have some kind of caching capabilities (like `String` class) but only if we use the `valueOf(primitive or string)` method to create the Wrapper class, this method take an `int` or `String` argument and returns the corresponding `WrapperClass`, example :

```
Integer price = Integer.valueOf(10_000);
Integer paw = Integer.valueOf("1");
Integer paw = Integer.valueOf("10_000"); // RUNTIME ERROR
```

Note that `valueOf()` and `intValue()` (for Integer) methods are less used in java because `autoboxing` has removed the need to use them.

The parse method

Wrapper classes also provide a method called `parsePrimitive(String)`, which returns **a primitive** from a String, example :

```
int price = Integer.parseInt("10000");
```

Note that the Character wrapper class doesn't have a parse method.

AUTOBOXING and UNBOXING

Since java 5, we can just assign a primitive value to a wrapper class, and java will convert it to the relevant wrapper class for us; This is called **autoboxing**. The reverse operation is called **unboxing**. Example :

```
Integer cars = 3; // autoboxing
int numberOfCars = cars; // unboxing

Float price = 10_000.0f;
float price = price;
```

Please be aware of some special cases when using autoboxing, example :

```
List<Integer> list = new ArrayList<>();
list.add(1);
list.add(2);
list.remove(1); // autoboxing is not performed here, instead this will call the
method remove(int index)
System.out.println(list); // will print : [1]
```

In the previous example autoboxing is not performed for integer 1, since there is a method `remove(int index)` that take an int into argument. If we to remove element with value 1 from the list we should force creating a Wrapper class : `list.remove(new Integer(1));`

Chapter 6: CLASSES AND METHODS

CLASS IN JAVA

INTRODUCING ACCESS CONTROL

one of the features of encapsulation encapsulation is that it links data to the code that manipulates it, another feature is `access control`. Through encapsulation, you can control what parts of a program can access the members of a class. How a member of a class can be accessed is controlled by `access modifiers` atteched to its declaration. Java provides 4 `access modifiers`:
1- `public` : the memeber can be accessed by any other code from anywhere. 2- `package default` : by default the member of a class is public within its own package. 3- `protected` : used in case of inheretence. 4- `private` : the member can only be accessed by other members of its class.
NOTE The modules feature added by JDK 9 can also impact accessibility.
Usually, you will want to *restrict access to the data members* of a class—allowing access only through methods. Also, there will be times when you will want to define methods that are private to a class.

INTRODUCING STATIC

When using `static` keyword to declare class members or methods of class, they can be accessed/called without the need to create an instance of the class. When using `static` on method there will be some restrictions:

- static method can only access other static method.
- static method can only use static variables.
- static method can't use `this` nor `super`. (since they are features that are specific to instance objects)

If you need to do some computation to inistialize a `static` variable, you should use `static {}` block that get executed only once.
To call a static variable or method we use the class name followed by the variable or method, exemple : `ClassName.staticMethod()`.

INTRODUCING FINAL

`final` is used on variable, method or class. When used on variable it prevent them from being modified (those are called constants). final variables must be intialized, that's possible in 2 ways:

- when it's declared.
- or within the constructor.

Using `final` on methods, prevent them from being overriding. Using `final` on classes, prevent them from being inhereted.

NESTED & INNER CLASSES

It's possible to declare class within another class, such classes are called inner classes.

`NESTED CLASSES` Are nested classes that are declared using the `static` keyword are called `NEESTED CLASSES`, they can access instance variable of the outer classes only through an object instance.
`INNER CLASSES` Are `non-static` nested classes, they have access to all instance members and methods of the outer class.
A nested class has access to the members, including private members, of the class in which it is nested. However, the enclosing class does not have access to the members of the nested class. An inner class can be instantiated only in the context of outer class, exemple `Outer.Inner innerClass = outerInstance.new Inner()`

```
public class Outer {

  private int outerVar;

  class Inner {
    private int innerVar;

    public void method() {
      innerVar = 20;
      outerVar = 30;
    }
  }

  public static void main(String[] args) {
    Outer p = new Outer();
    Inner i = p.new Inner(); // here inner class are used in the context of the outer
instance classe
    i.method();
    System.out.println("outerVar = " + p.outerVar); // this will display 30
    System.out.println("innerVar = " + i.innerVar); // this will display 20
  }
```

```
}
```

METHOD IN JAVA

ARGUMENT PASSING IN JAVA

In general, there are 2 ways that a computer language pass arguments to methods :

1- `Call-by-value`: copie the value of the parameter to a formal parameter of the subroutine;any change made to the parameter subroutine have no effect on the argument. 2- `Call-by-reference`: a reference to the argument (not the value) is passed to the method, any change made on the reference of the parameter will affect the argument used to call the subroutine.
Java use `call-by-value` for passing arguments.
Passing primitives to a method When you pass a primitive type to a method, it is passed by value. Thus, a copy of the argument is made, and what occurs to the parameter that receives the argument has no effect outside the method.

Example:

```
public static void doubleNumber(int a) {
    // this will not affect the paramater passed to the method
    a = a*2;
}
public static void main(String[] args) {
    int a = 4;
    doubleNumber(a);
    System.out.print("a = " + a); //This will print 4
}
```

Passing objects to a method When instanciating an object, java creates 2 things; a reference stocked in the `Stack` that references the Object value stocked in the `Heap`. Thus when you pass an object as an arguments to a method you're passing the reference of the object not its value, The parameter that will be created inside the subroutine will also refer to the same object since it's a copy of parameter which is a reference, so finally it's like you are using `Call-by-reference`. One should pay attention when passing objects to methods.
Example:

```
class Product {
    private String ref;
    private Double price;

    // constructors, getters and setters
}

public class Main {
```

```
    public static void doublePrice(Product p) {
        p.setPrice(p.getPrice()*2);
    }
    public static void main(String[] args) {
        var p= new Product("ref1", 100_000L);
        System.out.print("product price= " + p.getPrice()); //This will print
200_000L
    }
}
```

REMEMBER When an object reference is passed to a method, the reference itself is passed by use of call-by-value. However, since the value being passed refers to an object, the copy of that value will still refer to the same object that its corresponding argument does.

RECURSION

Recursion is the attribute that a method to call itself. a method that call itsvaelf is said to be `recurive`.
Exemple:

```
public int factorial(int n) {
    int result;
    if(n==1) return 1;
    result = fact(n-1) * n;
    return result;
}
```

A large number of recursive calls to a method could cause a stack overrun. Because storage for parameters and local variables is on the stack and each new call creates a new copy of these variables, it is possible that the stack could be exhausted. If this occurs, the Java run-time system will cause an exception. However, this is typically not an issue unless a recursive routine runs wild.
When a method calls itself, new local variables and parameters are allocated storage on the stack, and the method code is executed with these new variables from the start. As each recursive call returns, the old local variables and parameters are removed from the stack, and execution resumes at the point of the call inside the method. Recursive methods could be said to “telescope” out and back.

When writing recursive methods, you must have an if statement somewhere to force the method to return without the recursive call being executed. If you don’t do this, once you call the method, it will never return.

INHERTENCE

Definition

Inheritance is one of the cornerstones of object-oriented programming because it allows the creation of hierarchical classifications. Using inheritance, you can create a general class that defines traits common to a set of related items. This class can then be inherited by other, more specific classes, each adding those things that are unique to it. In the terminology of Java, a class that is inherited is called a superclass. The class that does the inheriting is called a subclass. Therefore, a subclass is a specialized version of a superclass. It inherits all of the members defined by the superclass and adds its own, unique elements.

How to

To inherit from a class you simply put the keyword `extends` after subclass declaration and before superclass name, example :

```
class A {
    // class members
}
class B extends A {
    // class members
}
```

When doing this, all protected and public variables and methods of class A will be in available in B. *private members of class A will not be accessible in class B*

Take away

- A subclass will inherit all members and method from superclass, except private ones.
- A class can only inherit from one class at a time. Java doesn't support the inheritance of multiple superclasses into a single subclass.
- A reference variable of a superclass can be assigned a reference to any subclass derived from that superclass. Exemple :

```
public class Main() {
    // we declare it as nested class, so that the creation of its instance will not
require the presence of the outer instance
   static class Duck {
       public boolean canFly() {
           return true;
       }
   }
   static class RubberDuck extends Duck {
       @Override
       public boolean canFly() {
```

```
            return false;
        }
    }
    static class Mallar extends Duck {
    }
    public static void main(String[] args) {
        Duck d = new Mallar();
        System.out.println(d.canFly()); // will display true
        d = new RubberDuck();
        System.out.println(d.canFly()); // will display false
    }
}
```

The use of SUPER

Whenever a subclass needs to refer to its immediate superclass, it can do so by use of the keyword `super`. `super` has 2 general forms: - The first calls the superclass' constructor. - The second is used to access a member of the superclass that has been hidden by a member of a subclass.

Using super to Call Superclass Constructors

A subclass can call a constructor defined by its superclass by use of the following form of super:

```
class SubClass extends SuperClass{
    private int b;
    class SubClass() {
        super(); // must be the first statment in the constructor declaration
        this.b = 20;
    }
}
```

EXERCISE NOTES

Foundation Test 1

Callable vs Runnable uses cases :

`Callable<T>` and `Runnable` are both functional interfaces that can be used to run asynchronous tasks in separate threads. The main difference between them is that :

Callable	Runnable
Has the method `T call() throws Exception;`	Has the method `void run();`
Is used for tasks that return a result	Is used when task doesn't return a result
Is used for tasks that throws checked exceptions	Is used for tasks that doesn't throws checked exceptions

`Callable<T>` and `Runnable` can be used in multiple ways:

- Callable using one of `ExecutorService` methods :

```
<T> Future<T> submit(Callable<T> task);
<T> T invokeAny(Collection<? extends Callable<T>> tasks);
<T> List<Future<T>> invokeAll(Collection<? extends Callable<T>> tasks) throws
InterruptedException;
```

- Runnable can be used to **directly** create a `Thread`, or via ExecutorService :

```
/**
 * Using Thread
 */
Thread t1 = new Thread(aRunnable);
// the above is equivalent to :
Thread t2 = new Thread(() -> {
    // do somthing
});

/**
 * Using ExecutorService
 */
<T> Future<T> submit(Runnable task, T result); // Submits a Runnable task for
execution and returns a Future containing the result
Future<?> submit(Runnable task); // submit a task and returns null on succeful
execution
```

Connection

When a connection is created, it is in `auto-commit` mode by default. This means that each individual SQL statement is treated as a transaction and is automatically committed right after it is completed. Thus **To enable a set of statement to be grouped in the same transaction we should disable the auto-commit mode**, right after creating the `Connection` calling `conn.setAutoCommit(false);`

Switch statement rules

Here are the rules for a switch statement: 1- Only String, byte, char, short, int, (and their wrapper classes Byte, Character, Short, and Integer), and enums can be used as types of a switch variable. (String is allowed only since Java 7). 2- The case constants must be assignable to the switch variable. For example, if your switch variable is of class String, your case labels must use Strings as well. 3- **The switch variable must be big enough to hold all the case constants**. For example, if the switch variable is of type char, then none of the case constants can be greater than 65535 because a char's range is from 0 to 65535. 4- All case labels should be COMPILE TIME CONSTANTS. 5- No two of the case constant expressions associated with a switch statement may have the same value. 6- At most one default label may be associated with the same switch statement.

Static nested class/interface

Note the difference between an inner class and a static nested class. Inner class means a NON STATIC class defined inside a class. Remember: 1- **A NESTED CLASS** is any class whose declaration occurs within the body of another class or interface. 2- A top level class is a class that is not a nested class. 3- **AN INNER CLASS** is a nested class that **IS NOT EXPLICITLY OR IMPLICITLY DECLARED STATIC**. 4- A class/interface defined inside an interface is **implicitly static**. 5- **NESTED CLASSES** doesn have the access to the outer class instance **since it's static**. 6- An inner class can extend it's outer class. 7- Member variables of the outer class can be accessibles to the inner class **ONLY IF THEY ARE NOT SHADOWED** in the inner class. 8- More than one inner class can be associated to the outer class.

Inheritence

Constructors and static initializers are not members and therefore are not inherited.

If statement

The only time we can the `=` operator in an `if statement` without having compile/runtime error is when both operands are booleans (since `if(a=b)` get the value that is assigned, thus `if(a=b)` is equivalent to `if(b)`).
The expression (a = b) does not compare the variables a and b, but rather assigns the value of b to the variable a. **The result of the expression is the value being assigned**. Since a and b are boolean variables, the value returned by the expression is also boolean. This allows the expressions to be used as the condition for an if-statement. if-

clause and the else-clause can have empty statements. Empty statement (i.e. just ;) is a valid statement. However, the following is illegal: if (true) else; because the if part doesn't contain any valid statement. (A statement cannot start with an else!) But the following is valid: if(true) if(false); because if(false); is a valid statement.

Literals : What does need a casting

- Conversion from short to char & from char to short (they don't have the same range).
- Conversion from float to int.
- conversion from double to long ?

Further, if you have a final variable and its value fits into a smaller type, then you can assign it without a cast because compiler already knows its value and realizes that it can fit into the smaller type. This is called **implicit narrowing** and is allowed between byte, int, char, and, short but not for long, float, and double.

```
byte b = 20;
final int i = 10;
b = i; // is OK, since i is final; constant

final float f = 10.0;
i = f; // will not compile, even after changing 10.0 to 10.0f
```

String manipulation

- `char charAt(int index)`, take int (or lower : char, short, byte), and return the char at the given position starting from 0. It can take up to length-1 as argument. if we pass an invalid index it throws an `IndexOutOfBoundsException` (not necessary a `java.lang.StringIndexOutOfBoundsException`), so pay attention to some question in the exam that could trick you, like :

`charAt` throws StringIndexOutOfBoundsException if passed a value higher than or equal to the length of the string (or less than 0).
It is not a valid option because the implementation is free to throw some other exception as long as it is an `IndexOutOfBoundsException`.

Functional interface

Is an interface that contains exaclty **ONE `abstract` method**, and it may contain zero or more default methods and/or static methods in addition to the abstract method. **Functional interfaces** are usefull in the context of lambda expression, since a

functional interface contains exactly one abstract method, you can omit the name of that method when you implement it using a lambda expression.

Scope restriction

The order of scope restriction is as follow : public < protected < package (or default) < private. (here, public is least restrictive and private is most restrictive).