OCA Summary : Chapter 1 : Java Basics

The structure of a Java class and source code file

- If a class includes a `package` statement, all the `import` statements should follow it.
- A Java source code file (.java file) can define multiple classes and interfaces.
- A `public` class can be defined only in a source code file with the same name.
- `package` and `import` statements apply to all the classes and interfaces defined in the same source code file (.java file).
- `System.lang` & package with no name (if there's no package declaration) are by default imported.

Executable Java applications

- For a class to be executable, the class should define a `main` method with the signature `public static void main(String args[]` *or* `String... varargs` *or* `String[] amines)`. The positions of `static` and `public` can be interchanged.
- A class can define multiple methods with the name `main`, provided that the signature of these methods doesn't match the signature of the `main` method defined in the previous point. These other methods with different signatures aren't considered *the* `main` method.

Java packages

- You can import either a single member or all members (classes and interfaces) of a package using the `import` statement.
- You can't import classes from a subpackage by using the wildcard character, an asterisk (*), in the `import` statement.
- A class from a default package can't be used in any named packaged class, regardless of whether it's defined within the same directory or not.
- ✓ `import static <package>.<class>.*;` × `import static <package>.<class>;`
  ✓ `import static <package>.<class>.<field>;` × `import static <package>.*;`
- The members of default packages are accessible only to classes or interfaces defined in the same directory on your system.
- The syntax to import `static method()` : `import static <package>.<class>.method`

Java modifiers

Access modifiers	Package Classes	Non-package Classes	Nested Classes
public	✓	✓	✓
protected	✓	*if derived*	✓
default	✓	×	✓
private	×	×	✓

Non-access modifiers	Interface	Class	Variable	Method
abstract	*redundant*	*abstract/concrete methods*	×	*no body can't be static*
final	×	*cannot be extended*	*1 assignement possible*	*cannot be overriden*
static	*only nested*		*common to all instances access only static members*	
strictfp	✓	✓	✓	✓ *(none-abstract)*
synchronized	×	×	×	✓
native	×	×	×	✓
transient	×	×	✓	×
volatile	×	×	✓ *(none- local)*	×

- `protected` and `private` are not valid modifiers for a class or interface definition.
- A `private` method cannot be `abstract` ■ A return type must be directly before the method's name.

Chapter 2 : Working with Java data types

Primitive data types

■ Primitive data types: `char`, `byte`, `short`, `int`, `long`, `float`, `double`, and `boolean`.

Primitive wrappers

■ `PrimitiveWrapper.MIN_VALUE + 1 == PrimitiveWrapper.MAX_VALUE.`
■ Two primitive wrappers are never compatible (ex : `Integer` & `Long`).
■ Primitive wrappers are immutable. When trying to modify a primitive wrapper value, a new instance is created and the original instance remains unchanged.
■ A primitive wrapper constructor's argument accepts only a compatible primitive value.
■ A primitive wrapper can be compared to a compatible primitive using the == operator.
■ A primitive wrapper's constructor accepts either the primitive value or the String representation.
■ `compareTo(Wrapper)` compares two wrappers of the same type.
■ `primitiveValue()` returns the primitive value of the conversion of the selected wrapper.
■ `static parsePrimitive(String)` returns the primitive value of the String (if it is valid).
■ `static decode(String)` returns the primitive wrapper value of the String (if it is valid).
■ `static valueOf(String/primitive)` returns the primitive wrapper value of the parameter.
■ `static toString(primitive)` returns the String value of the parameter
■ `toString()` returns the String value of the selected wrapper.
■ If a method is given a primitive argument and no method signature with a corresponding primitive type was found. The primitive is boxed into a wrapper to find a corresponding object type parameter method.
■ Java reuses all wraper objects whose values are between -128 and 127.

Numeric data types

■ The float and double data types use 32 and 64 bits, respectively, to store their values.
■ The default type of integers—that is, nondecimal numbers—is `int`.
■ To designate an integer literal value as a `long` value, add the suffix `L` or `l` to the literal value.
■ Prefixes : **octal :** 0 (zero), **hexadecimal :** 0x or 0X, **binary :** 0b or 0B.
■ Underscores within the Java literal values : **not in :** start, end, after binary or hexa prefixes, before suffixes (l, L, f, F, d, D) , adjacent to a point, string of digits..
■ The default type of a decimal number is `double`.
■ To designate a decimal literal value as a `float` value, add the suffix `F` or `f` to the literal value.
■ The suffixes `D` and `d` can be used to mark a literal value as a `double` value. Though it's allowed, doing so is not required because the default value of decimal literals is `double`.
■ Decimal types accept all integer types (including `char`). `char` accepts only litterals and `char`.
■ Integers accept `char` litterals depending on their capacity, but only `int` & `long` accepts a `char` variable.
■ A method can't accept an argument that is a literral without casting it if the parameter is an integer different than `int` or `long`.
■ When comparing a decimal to an integer, the decimal is casted to it's integer value.
■ There is a loss of precision when casting to a `float` or `double`.
■ All operands of type byte, char or short are promoted AT LEAST to an int before math operations.
■ CONSTANT values up to `int` can be assigned to variables of lesser size if it fits.

Character primitive data types

■ A `char` data type can store a single 16-bit Unicode character; that is, it can store characters from virtually all the world's existing scripts and languages.
■ You can use values from `'\u0000'` (or 0) to a maximum of `'\uffff'` (or 65,535 inclusive) to store a `char`. Unicode values are defined in the hexadecimal number system.
■ Internally, the `char` data type is stored as an unsigned integer value (only positive integers).
■ When you assign a letter to a `char`, Java stores its integer equivalent value. You may assign a positive integer value to a `char` instead of a letter, such as 122.

Valid identifiers

- A valid identifier starts with a letter, a currency sign, or an underscore.
- A valid identifier can't contain any special characters, including `!`, `@`, #, `%`, `^`, `&`, `*`, `(`, `)`, `'`, `:`, `;`, `[`, `/`, \, or `}`.
- Keywords : assert, goto, const, native, strictfp, volatile, instanceof, default, null.

Operators

Precedence		
Postfix	`a++, a--, [], .member, method(),`	Left to Right
Unary	`++a, --a, !, ~`	Right to Left
new & Cast	`new, (cast)`	Right to Left
Multiplication	`*, /, %`	Left to Right
Addition	`+, -`	Left to Right
Relational	`<, >, <=, >=`	Left to Right
Equality	`==, !=`	Left to Right
Logical AND	`&&`	Left to Right
Logical OR	`||`	Left to Right
Conditional	`?:`	Right to Left
Assignement	`=, +=, -=, *=, /=, %=`	Right to Left

- All compound assignment operators internally do an explicit cast.

Chapter 3 : Methods and encapsulation

Scope of variables

■ In a method, if a local variable exists with the same name as an instance variable, the local variable takes precedence.
■ If a reference variable A can never reference a variable of type B : `A instanceof B` is a compilation error. Otherwise it is not a compilation error but it could be a runtime exception.
■ A `static` method can't be `abstract`.
■ An unprecedeed semicolon is allowed.

Object's lifecycle

■ Varargs : `int... days` : only one and the last one in parameters list.
■ If there is code that can be executed only after a `return` statement, the class will fail to compile.

Create an overloaded method

■ It is a compilation error when calling an overloaded method that accepts literal values that may correspond to 2 different types (**example :** `double min(double a, int b); double min(int a, double b); double x = min(2,3);`).
■ For Objects & primitives, when a method is called it's parameters are expanded to the exact match prior to their nearest type. For primitives : widening is preferred to wrappers, wrappers are preferred to varargs.

Constructors of a class

■ Default constructors are defined by Java, but only if the developer doesn't define any constructor in a class.
■ Constructors are NEVER inherited.
■ You can define a constructor using the four access modifiers: `public`, `protected`, default, and `private`.
■ If you define a return type for a constructor, it'll no longer be treated like a constructor.
■ An *initializer block* is defined within a class, not as a part of a method. It executes for every object that's created for a class.
■ If you define both an initializer and a constructor for a class, both of these will execute. The initializer block will execute prior to the constructor.
■ Unlike constructors, an initializer block can't accept method parameters.
■ An initializer block can create local variables. It can access and assign values to instance and static variables. It can call methods and define loops, conditional statements, and `try-catch-finally` blocks.
■ Constructors can't exist without a body.
■ Constructors accept all but only access modifiers.
■ Constructors may end with a semi-colon (ignored by the compiler).
■ The access type of a default constructor is same as the access type of the class.
■ Order of precedence when instantiating a class : `static parent, static child, init parent, constructor parent, init child, constructor child.`
■ An initializer (or `static`) block is called when a member (non-constant) or a constructor declared in the class (rather than inherited) is invoked, used or assigned. It's not called by only declaring a reference variable.
■ It is an obligation to initialize a `final` instance variable either in a constructor or in an initializer block.
■ Unlike non-final instance variables, `final` instance variables must be initialized before use.

Accessing object fields

■ The terms *encapsulation* and *information hiding* are also used interchangeably.
■ One of the best ways to define a well-encapsulated class is to define its instance variables as private variables and allow access to these variables using methods.

Chapter 4 : String, StringBuilder, Arrays, ArrayList

String & StringBuilder

■ A `String` object can be created by using the operator `new`, by using the assignment operator (=), or by using double quotes.
■ `String` objects created using the assignment operator are placed in a *pool* of `String` objects.
■ `String` objects created using the operator `new` are never placed in the pool of `String` objects.
■ if not overloaded the method `toString()`of any object returns : object's name + `hashCode()`.
■ StringBuffer ⇔ synchronized StringBuilder.

	method	String	StringBuilder	Parameters	Mod
1	charAt	✓	✓	(int index)	×
2	substring	✓	✓	(int start) (int start, int end) returns String	×
3	subSequence	✓	✓	(int start, int end) returns CharSequence	×
4	length	✓	✓	( )	×
1	indexOf	✓	×	(char) (String) (char/String, int start)	×
2	trim	✓	×	( )	×
3	startsWith	✓	×	(String) (String, int startOrig)	×
4	endsWith	✓	×	(String)	×
5	replace	✓	×	(char old, char new) (CharSequence old, CharSequence new)	×
6	intern	✓	×	() returns reference of the string in the pool	×
7	equalsIgnoreCase	✓	×	()	×
8	concat	✓	×	(String new)	×
1	new StringBuilder	×	✓	( )16 chars, (String) +16 chars, (CharSequence) (int capacity)	✓
2	append	×	✓	add at the end: (type) (CharSequence,start,nb)	✓
3	insert	×	✓	(startOrig, CharSequence, startNew, nbNew) (startOrig, type)	✓
4	delete	×	✓	(int start, int end)	✓
5	deleteCharAt	×	✓	(int pos)	✓
6	reverse	×	✓	( )	✓
7	replace	×	✓	(int start, int end, String new)	✓
8	setLength	×	✓	(int) appends 0s at the end if larger	✓
9	ensureCapacity	×	✓	(int minimum) sets capacity if less	
0	indexOf	×	✓	(String) (String, int start)	×

*end not included

Arrays

■ An array's size only accepts byte, char, short, int. It is allowed to be equal to a 0.
■ The creation of an array involves three steps: declaration of an array, allocation of an array, and initialization of array elements.
■ Square brackets can follow either the variable name or its type and they should be empty. In the case of multidimensional arrays, it can follow both of them.
■ `int T[]=new int[5];`✓ `int T[][]={{0,1},{3,4,5}};`✓
`int T[][]=new int[×][×]{{0,1},{3,4,5}};`✓ `int T[]; T=new int[×]{0,1};`✓
`int T[]; T={0,1};`× `int[] multiArr[]; multiArr = new int[2][];`✓
■ The Java compiler doesn't check the range of the index positions at which you try to access an array element.
■ All the arrays are objects and can access the variable `length`, which specifies the number or components stored by the array.
■ For primitives, an array of a certain type can't refer to an array of a compatible type (Ex : int[] array = new char[5] ×). But for objects it's possible (Ex : Animal[] array = new Lion[10] ✓).
■ `public static void arraycopy(Object src, int srcPos, Object dest, int destPos, int length)` (`System.arraycopy`)

ArrayList

method	*Parameters*
add	(object) (i, object)
set	(i, object)
remove	(i) (object) first occurence, test of equality with equals()
addAll	(Collection) (i, Collection)
clear, size, clone, toArray	( )
get	(i)
contains, indexOf, lastIndexOf	true, false (object) index, -1

■ By default, an `ArrayList` creates an array with an initial size of 10 to store its elements.
■ Unlike arrays, you don't have specify an initial size to create an `ArrayList`.
■ If you frequently add elements to an `ArrayList`, specifying a larger capacity will improve the code efficiency.
■ `ArrayList` allows `null` values to be added to it.
■ `ArrayList` supports generics, making it type safe.
■ Internally, an array of type `java.lang.Object` is used to store the data in an `ArrayList`.
■ To access the elements of an `ArrayList`, you can use either the enhanced `for` loop, `Iterator`, or `ListIterator` :

```
ListIterator<String> iterator = myArrList.listIterator();
While(iterator.hasNext()) ...iterator.next()...
```

■ An iterator (`Iterator` or `ListIterator`) lets you remove elements as you iterate through an `ArrayList`. It's not possible to remove elements from an `ArrayList` while iterating through it using a `for` loop.
■ `ArrayList` has subclasses such as `AttributeList`, `RoleList` & `RoleUnresolvedList`.
■ An `ArrayList` can only store Objects.
■ `ArrayList` implements `RandomAccess` interface → direct access → constant time access.
■ `ArrayList<E>` extends `AbstractList<E>` extends `AbstractCollection<E>`.
■ `ArrayList<E>` implements `Serializable`, `Cloneable`, `List<E>`, `RandomAccess.`
■ `List<E>` extends `Collection<E>`, `Iterable<E>.`

Comparing objects for equality

■ Except for String, The default implementation of the `equals` method only compares whether two object variables refer to the same object.
■ Java equals contract : `public boolean equals(object o)` : reflexive, symmetric, transitive, consistent, same class, `if(x !=null) x.equals(null) returns false.`

Chapter 5 : Flow Control

switch statements

■ A `switch` accepts only char, byte, short, int, String (and Enums) as a variable or expression.
■ The `case` value should be a compile-time constant (`final` and assigned at declaration or literrals), assignable to the argument passed to the `switch` statement.
■ The `case` value can't be the literal value `null`.
■ The `case` values cannot be repeated.
■ A `switch` head accepts a constant.
■ A `switch` must have a body but could be empty. There could be nothing after a `case` label as well.

for loops

■ The definition of any of the three `for` statements—initialization statements, termination condition, and update clause—is optional. For example, `for (;;);` and `for (;;){}` are valid code for defining a `for` loop (infinite loops).
■ A `for` loop can declare and initialize multiple variables in its initialization block, but the variables that it declares should be of the same type.
■ The update clause of a `for` loop may contain only : increment/decrement expression, method invocation or class instance creation expression.

Enhanced for loops

■ The enhanced `for` loop can't be used to initialize an array and modify or delete its elements.
■ The enhanced `for` loop elements are passed by value.
■ The only valid modifier for the enhanced `for` loop iterator is `final`.

Labeled statements

■ You can add labels (`label:`) to a code block defined using braces, `{}`, all looping statements (`for`, enhanced `for` loop, `while`, `do-while`), conditional constructs (`if` and `switch` statements), expressions and assignments, `return` statements, `try` blocks, `throws` statements or method calls.
■ You can't add labels to declarations of variables.
■ If there is no brace `{` after the label. Only one expression is included in the label's area.
■ You can use a labeled `break label;` statement to exit an outer loop or any kind of block.
■ You can use a labeled `continue label;` statement to skip the iteration of the outer loop.
■ If not inside a loop, you can't use a `break` or a `continue` statement inside an `if` or `else` block.
■ `break/continue label;` must be used inside the block that `label` refers to.
■ A `continue` statement can be used only in loops.

while & do while loops

■ The condition expression in a `while` & `do while` header is required.
■ It is a compilation error to put `false` as a condition test in a `while` or `for` header (block unreachable).

Conditions

■ It is not a compilation error to put a `false` as a condition test in an `if` header.
■ For the operator `?:` both sides of `:` must return the same type (except void).

Chapter 6 : Working with inheritance

Inheritance with interfaces and classes

■ An interface can extend zero or more interfaces. An interface cannot extend a class or implement an interface.
■ An interface's variables are implicitly `public`, `final` & `static`. They should be assigned at declaration. It's methods are implicitly `public` and they can't be `final` or `static`.
■ An `abstract` class can inherit a concrete class, and a concrete class can inherit an `abstract` class.
■ A base class can use reference variables and objects of its derived classes.
■ The keyword `extends` should always come before `implements`.
■ An interface can only be instantiated with an anonymous inner class.

Using super and this to access objects and constructors

■ The reference variable `super` can be used to access a variable or method from the base class if there is a clash of these names. This situation normally occurs when a derived class defines variables and methods with the same names as in the base class.
■ The reference variable `this` can be used to access the inherited members of a base class in the derived class.
■ When defining a class that has no constructors, if Java doesn't find a no-argument constructor in the base class it fails to compile.
■ The reference variables `this` and `super` can't be used in a static context.

Polymorphism with classes

■ The return type of the overriding method, if it is an Object, can be the same, or a subclass of the return type of the overridden method in the base class. But should be exactly the same if it is a primitive.
■ Classes are not polymorhpic if they don't define polymorphic methods, even if they extend each other.
■ Access modifiers for an overriding method can be the same or less restrictive but can't be more restrictive than the method being overridden.
■ With inheritance, the instance variables, exceptions & static members bind at compile time while instance methods bind at runtime.
■ The `hashcode()` and `equals()` methods can be overriden.
■ The overriding method may declare only exceptions declared in the original method or it's subclasses. It may choose not to declare any exception as well. (Checked exceptions).
■ The constructor of a subclass doesn't have to declare the exceptions declared in the called superclass constructor. If it declares an exception it can't be the subclass of the superclass's constructor exception.
■ Variables & `static` methods are not overriden, they are shadowed.
■ There is no way to go more than one level up for non-static methods.
■ Implementing two interfaces that have the same variable name X is no problem. Referring to X is a compilation error.
■ An `abstract` method can override a concrete method.
■ A `static` method cannot override a non-static method and vice-versa.

Nested classes

■ Nested classes that are declared `static` are called static nested classes. Non-static nested classes are called inner classes.
■ A nested class can be declared `private`, `public`, `protected`, or package private.
■ Static nested classes are accessed using the enclosing class name: `OuterClass.StaticNestedClas`
■ An inner class cannot define any static members itself.
■ An instance of InnerClass can exist only within an instance of OuterClass and has direct access to the methods and fields of its enclosing instance.
■ To instantiate an inner class, you must first instantiate the outer class. Then, create the inner object within the outer object with this syntax:

```
OuterClass.InnerClass innerObject = outerObject.new InnerClass();
```

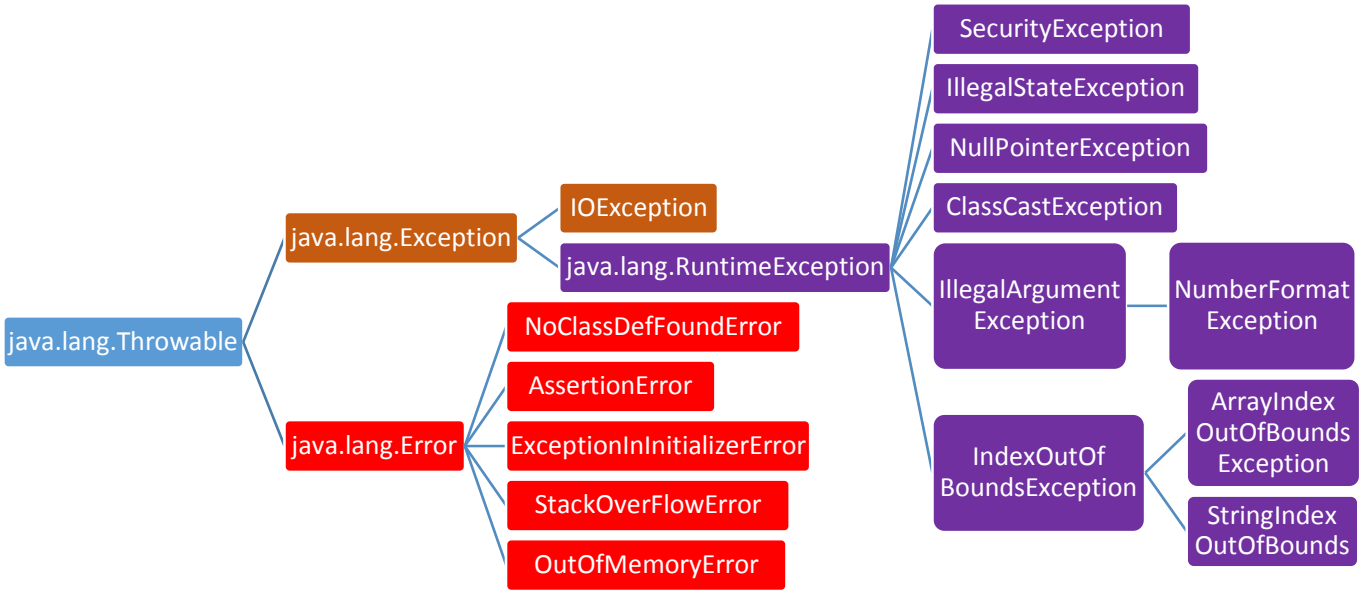
■ To access a shadowed variable `x` of the OuterClass in it's InnerClass : `OuterClass.this.x`

Chapter 7 : Exception handling

An exception is thrown

■ When a piece of code hits an obstacle in the form of an exceptional condition, it creates an object of subclass `java.lang.Throwable`, initializes it with the necessary information and hands it over to the JVM.
■ A `finally` block will execute even if a `try` or `catch` block defines a `return` statement.
■ If both `catch` and `finally` blocks define `return` statements, the calling method will receive the value from the `finally` block.
■ If a `catch` block returns a primitive data type, a `finally` block can't modify the value being returned by it. If a `catch` block returns an object, a `finally` block can modify the value being returned by it.
■ The `finally` block can't appear before a `catch` block.
■ The `finally` block always executes except if there is a `System.exit()` call.

Categories of exceptions



■ If a method calls another method that may throw a checked exception, either it must be enclosed within a `try-catch` block or the method should declare this exception to be thrown in its method signature.
■ You can use the operator `instanceof` to verify whether an object can be cast to another class before casting it.
■ Runtime exceptions arising from any of the following may throw `ExceptionInInitializerError`:
– Execution of an anonymous `static` block
– Initialization of a `static` variable
– Execution of a `static` method (called from either of the previous two items)
■ The error `ExceptionInInitializerError` can be thrown only by an object of a runtime exception.
■ `NoClassDefFoundError` is thrown by the JVM or a `ClassLoader` when it is unable to load the definition of a class required to create an object of the class.
■ Local variables can't be used without being initialized or assigned (compilation error if not guaranteed). But fields (object's) and elements (array's) are implicitly initialized to default values.
■ A `NullPointerException` will be thrown if the JVM finds a `null` exception in the catch header.
■ When `printStackTrace()` is called, implicitly (`throwable` not catched) or explecitly, it prints a complete chain of methods names and lines numbers. When `System.out.print(throwable)` is called only the method's name and the message are printed.
■ `AssertionError` is used when you reach code that should never execute.
■ Advantages of exceptions : Separating Error-Handling Code from "Regular" Code + Propagating Errors Up the Call Stack + Grouping and Differentiating Error Types.
■ If the JVM doesn't find *the main* method it throws `NoSuchMethodError`.