1Z0-808 Preparation

Java Basics

- Define the scope of variables
  - Class variables
  - Instance variables
  - Local variables ⚠ they must be initialized otherwise the code cannot be compiled
- Define the structure of a Java class
- Create executable Java applications with a main method; run a Java program from the command line; produce console output
- Import other Java packages to make them accessible in your code
  - Naming conflict rules:
    - Cannot import both java.util.* and java.sql.* because a class named exists in both packages => compilation error
    - When explicitly adding java.util.Date the code would compile
- Compare and contrast the features and components of Java such as: platform independence, object orientation, encapsulation, etc.
- Variables names
  - Authorized characters: letters, numbers, _, $
  - Cannot start with a number
- Numeric literals can contain
  - 0-9
  - 0X, 0x prefixes for **hexadecimal** (ex: 0X10 => 16)
  - 0B or 0b prefixes for **binary** (ex: 0b101 => 5)
  - Prefix with 0 for **octal**: int i = 053 means 43 in decimal
  - _ when surrounded by two digits
- Conversions
  - Not supported:
    - Conversion to a larger type: Int a = 8L, float f =8L, float f=1d;
  - Supported
    - Int/long/float/double a = 0x10 or 0x10
    - Int/long/float/double a = 0b111 or 0B111
    - Float f = 4L ;
- Garbage collector
  - Signature of finalize method: **protected void finalize()**
  - Garbage collection in not guaranteed to run even when calling **System.gc()**
- A method can have the same name as its class thus the same name as the constructor. It is distinguished by the fact of having a return type

Using Operators and Decision Constructs

- Use Java operators; use parentheses to override operator precedence
    - Operator precedence :
        - Post-unary operators expression++, expression--
        - Pre-unary operators ++expression, --expression
        - Other unary operators +, -, !
        - Multiplication/Division/Modulus *, /, %
        - Addition/Subtraction +, -
        - Shift operators <<, >>, >>>
        - Relational operators <, >, <=, >=, instanceof
        - Equal to/not equal to ==, !=
        - Logical operators &, ^, |
        - Short-circuit logical operators &&, ||
        - Ternary operators boolean expression ? expression1 : expression2
        - Assignment operators =, +=, -=, *=, /=, %=, &=, ^=, |=, <<=, >>=, >>>=
    - Numeric promotion in arithmetic binary operation
        - Smaller data types, namely **byte**, **short**, and **char**, are first promoted to int any time they're used with a Java **binary** arithmetic operator, even if neither of the operands is int.
        - 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.
        - Binary numeric promotion is performed on the operands of certain operators:
            - The multiplicative operators *, /, and %
            - The addition and subtraction operators for numeric types + and -
            - The numerical comparison operators <, <=, >, and >=
            - The numerical equality operators == and !=
            - The integer bitwise operators &, ^, and |
            - In certain cases, the conditional operator ? :
- Test equality between Strings and other objects using == and **equals()**
    - Remember that literals are pooled. "TOTO" == new String("TOTO") is false
    - instance1.**equals**(instance2) will return false if the equals method is not implemented by the belonging class.
- Create if and if/else and ternary constructs
- Use a switch statement
    - Are supported only:
        - integer numerics (byte, short, int) and their wrapping types
        - char and Character
        - Strings
        - Enums
    - ⚠ All the case values must be known at compile time(final or literals)
    - ⚠ Long and long are NOT supported
    - No duplicate case
- Logical operators:
    - ^ is for exclusive => a^b returns true if **one and only** one of a and b is true
    - | is for or => a | b returns true if one or both of a and b is true
    - & is for and => a & b returns true if both a and b are true

Using Loop Constructs

- Create and use while loops
  - ⚠ Take care of unreachable code that could cause compilation errors:
    - **while** (false) { … } would not compile
    - **while** (1==2) { … } would not compile
    - **while** (booleanVariable = false) would compile
- Create and use for loops including the enhanced for loop
  - Classic version: **for**(**initStatement**; **booleanExpression**, **updateStatements**) { **body** }
    - The clauses are evaluated in the following order:
      1. Initialization statement executes
      2. If booleanExpression is true continue, else exit loop
      3. Body executes
      4. Execute **updateStatements**
      5. Return to Step 2
    - ⚠ The updateStatements are executed after the body
    - All the three parts are optional
    - ⚠ It is possible to have multiple increment statements separated by “,”
- Create and use do/while loops
- Compare loop constructs
- Use break and continue

Working with Inheritance

- Describe inheritance and its benefits
- Develop code that makes use of polymorphism; develop code that overrides methods; differentiate between the type of a reference and the type of an object
  - Overriding methods must comply with the following rules:
    - The access to the overridden method in the child cannot be more restrictive than the parent
    - The return type must be of the same type or of a subtype of return type of the parent method
    - The signature must be the same (name + parameters)
    - The exception being thrown must not be new or wider than the exception thrown by the parent
- Determine when casting is necessary
- Use super and this to access objects and constructors
  - Call to super and this must be the first one if placed in a constructor
  - By default if no call to super is done, the compiler automatically adds a call to super()
    - ⚠ If the parent class does not have a no argument constructor, the child class must explicitly call super with arguments otherwise with a compilation error occurs
- Use abstract classes and interfaces
  - Abstract classes
  - Interfaces
    - An interface is by definition abstract so the word abstract can be omitted
    - An interface cannot be private, protected or final => **package private** or **public**
    - An interface can extend another interface but not implement it
    - A class can implement an interface but not extends it
    - Variables of an interface are assumed to be **public static final**
    - Default methods
      - Must provide an implementation
      - Must be public
      - Cannot be final
      - ⚠ Cannot override methods from Object class

        - Static methods
            - ⚠ Cannot be **abstract**
            - Must provide an implemention
        - A class cannot implement two interfaces that have the same **default** methods signatures
            - ⚠ Except if it overrides this method
- Execution order
    - Static blocks
    - Ancestors Constructor
    - Anonymous blocks
    - Constructor

Working with Selected classes from the Java API

- Manipulate data using the StringBuilder class and its methods
    - ⚠ StringBuilder is NOT String: StringBuilder sb= “string” does not compile
    - Important methods
        - **append**() modifies the content of the object
        - **charAt**(), **indexOf**(), **substring**() and **length**() have the same behavior as String (the content is not modified)
        - **insert**(index, String) inserts a new string at specified index
        - **delete**(startIndex, endIndex) deletes the characters comprised between startIndex exclusive and endIndex exclusive.
        - **deleteCharAt**()
        - **reverse**()
        - **capacity**() returns the size of the internal buffer
        - **getChars**(start, end, dest, destStart) copies the chars of this StringBuilder from index **start** to index **end (exclusive)** into dest (char[]) starting at index destStart. An **ArryIndexOutOfBoundsException** may be thrown
    - Capacity of a StringBuilder:
        - Can be set using the constructor **StringBuilder**(int capacity)
        - All the others constructors will reserve 16 characters more than needed
        - If the no argument constructor is used the capacity will be 16
- Create and manipulate Strings
    - String as many constructors including:
        - String() no argument constructor
        - String(“<my string>”)
        - String(char[]), String(char[], offset, amount)
        - String(byte[]), String(byte[], offset, amount)
        - String(StringBuilder), String(StringBuffer)
    - Concatenation obey the following rules:
        - If both operands are numeric, + means numeric addition.
        - If either operand is a String, + means concatenation.
        - The expression is evaluated left to right.
    - Strings are Immutable: they can’t be modified once their value is set.
    - The **string pool** is an area of memory where the JVM stores the String literals as a consequence, there is a difference between
        - String s = “TOTO” // it goes in the String pool
        - String s = new String (“TOTO”) // it does not go into the StringPool and can be garbage collected.
    - The numbering of characters positions starts with 0
    - ⚠ String cannot be created from the addition of two numbers: String s = int1 + int2 will not compile.
    - Importants points on some methods:

        - **substring**(n,m) returns the sequence of characters comprised between the nth inclusive and the mth exclusive
        - **trim**() does not only remove spaces but also **\t**, **\r** and **\n** at the end and the beginning
        - ⚠ **compareTo** is based on the unicode value of each character.
- Create and manipulate calendar data using classes from java.time.LocalDateTime, java.time.LocalDate, java.time.LocalTime, java.time.format.DateTimeFormatter, java.time.Period
    - LocalDate can be created the following ways
        - LocalDate ld= LocalDate.**now**() // Current date
        - LocalDate ld = LocalDate.**of**(1970, 10, 27)
        - LocalDate ld = LocalDate.**of**(1970, Month.OCTOBER, 27)
    - LocalTime can be created using
        - LocalTime lt = LocalTime.**now**() ;
        - LocalTime lt = LocalTime.**of**(int hour, int minute)
        - LocalTime lt = LocalTime.**of**(int hour, int minute, int seconds)
        - LocalTime lt = LocalTime.**of**(int hour, int minute, int seconds, int nanos)
    - LocalDateTime follows the same logic
        - LocalDateTime ldt = LocalDateTime.**now**()
        - LocalDateTime ldt = LocalDateTime.**of**(ld, lt)
        - LocalDateTime ldt = LocalDateTime.**of**(1970, 10, 27, 22,15)
        - LocalDateTime ldt = LocalDateTime.**of**(1970, Month.OCTOBER, 27, 22, 15, 23, 03)
        - In addition it is possible to obtain a LocalDateTime from LocalDate using method **atTime**() which allows adding time
    - Period materialize a period of time in year, month and days
        - Period.**of**(int, int, int)
        - Period.**ofYears**, Period.**ofMonths**, Period.**ofWeeks**, Period.**ofDays**
        - When printed the string representation will conform to the pattern **PxYyMzM**. If one of the fields is 0 it is not printed.
    - Manipulation of these new types very easy
        - **plusXXX** adds a period of time
        - **minusXXX** removes a period of time
        - **XXX** goes from **Years** to **Nanos**
          ex : LocalDateTime.**of**(1970, 10, 27, 22,15).plusMinutes(45)
    - General important considerations
        - ⚠ these types are **immutable** so they are not altered when using plus or minus
        - ⚠ to create objects of one of these types only static methods can be used. The constructors for each one of these classes is private and can't be invoked.
        - DateTimeException may be thrown in case of inconsistent dates ➔ ex: LocalDate.of(2019, 2, 29)
- Declare and use an **ArrayList** of a given type
    - Can be declared with or without <>
    - Duplicates are allowed
    - The important methods are:
        - **boolean add(E)** returns true if the element has been added false otherwise
            - add(E) adds an element at the end of the array
        - **void add**(index, E) adds E at a specific index
        - **boolean remove** returns true if a match is found and removed false otherwise
            - remove(E) remove the first element that matches
        - **E remove(index)** removes an element at a specific index
            - It returns the removed object
        - **E set(int, E)** replace element at specified index and returned the replaced element
        - **isEmpty**() and **size**()
        - **clear**()
        - **contains**()

            - **equals**() returns true if the two lists have the same elements in the same order. Calls the method equals on each element
    - Conversions
        - ArrayList can be converted to array using **toArray**
            - By default **toArray** returns an array of objects: Object[]
            - To return an array of a specific type, set as a parameter an array of the desired type in this case:
                - If the specified array can contain the list, it is returned
                - If not because of its insufficient size, a new one is returned
        - An array can be converted to an ArrayList using **Arrays.asList**()
            - ⚠ In this case the original array is linked to the ArrayList if a change is made to one, the modification is done to the other.
        - Diamond operator considerations
            - ⚠ If the type is not provided the compiler will consider elements of the list as Objects
- Write a simple Lambda expression that consumes a Lambda Predicate expression
- Functional programming
    - Functional interfaces must
        - Have one and only one abstract method
        - It can also have default and static methods

Working With Java Data Types

- Declare and initialize variables (including casting of primitive data types)
    - Default data types
        - Defaults are int and double
        - For primitive types whidenning is supported: long l = 4 is similar to long l = 4L
        - For wrapped classes, whidenning is not supported with autoboxing: Long L = 4 would not compile.
- Differentiate between object reference variables and primitive variables
- Know how to read or write to object fields
- Explain an Object's Lifecycle (creation, "dereference by reassignment" and garbage collection)
- Develop code that uses wrapper classes such as Boolean, Double, and Integer
    - ⚠ when creating a boolean from a String any value other than “**true**” would give the value false
      Ex: Boolean bool = new Boolean(“True’).booleanValue() ➔ false

Creating and Using Arrays

- Declare, instantiate, initialize and use a one-dimensional array
    - Declaration: int[] intArr or int intArr[] can be used to declare array
    - Instantiation:
        - T[] arr = new T[size]   ⚠ size is of type int
    - Initialization
        - int[] intArr = new int[4]
        - int[] intArr = {2,4,3,5}
        - ⚠ declarations on the same line:
            - int intArr[], oneInt means that we declare one array of int and one int
            - int [] intArr, oneInt ; means that we are declaring 2 arrays
    - ⚠ be careful when casting
        - Cannot cast int[] to double[]
    - The java.util.Arrays class offers some useful static methods
        - **binarySearch**(sortedArray, element) return the index of element item in sortedArray

            - ⚠ be careful with method binarySearch for unsorted arrays (binarySearch applies a dichotomy). The result is unpredictable otherwise
            - Returns -1 if the element is not found
        - **Arrays.asList**(T...t) converts a list of T or an array of T into a List<T>.
        - **sort**(array) and **sort**(array, from, to) will sort the whole array or just from index “from” inclusive to index “to” exclusive.
        - **equals**(arr1, arr2), returns true if both arrays contains the same elements in the same order.
            - If both are null they are considered as equal
        - **deepEquals**(Object[] arr1, object[] arr2)
            - Works with multidimensional arrays
        - **copyOf**(int[] a, int newLength) returns a copy of a with size newLength
    - When passing an array as a parameter of a method, the form { a, b, c } cannot be used. Instead, it should be passed as follows: new T[] { a, b, c }
- Declare, instantiate, initialize and use multi-dimensional arrays
    - More or less same logic as one-dimensional arrays with more variants:
      Int[] numbers3D[][], number2D[]...
    - Particularities:
        - A multidimensional array should be thought of as an array of arrays where all the elements are not constrained to have the same number of items

Working with Methods and Encapsulation

- Create methods with arguments and return values; including overloaded methods
  - The following rules applies to varargs:
    - Only one vararg per method signature
    - If a vararg is present it must be the last argument
    - When invoking a method with a vararg, the syntax allow passing an array or a list of values.
    - ⚠ Take care when overloading: a vararg is considered as an array, therefore it **method(int[] x)** is considered to be the same signature as **method(int... x).**
- Apply the static keyword to methods and fields
  - ⚠ it is possible to call a static method from an instance even if it is valued as null
  - Static variables must be initialized
    - On declaration
    - In static block
    - In default constructor
  - Static imports are used to import class variables and static method the syntax is
    - import static my.package.MyClass.* to import all static fields or methods
    - import static my.package.MyClass.myMethod to import only one method or field
    - ⚠ **static import** is not a correct syntax only **import static** is correct.
    - ⚠ static import does not replace import
    - It is not possible to do static import of methods or fields with the same name
- Create and overload constructors; differentiate between default and user defined constructors
  - An **overloading** method has the following characteristics:
    - Same name
    - Different parameter list or parameter types
    - May have different return types
    - Access level may be different
  - Overriding
- Apply access modifiers
  - For classes
    - For top level classes, only **public** and **default** are supported
    - For inner classes, **protected**, and **private** are also allowed
  - List of access modifiers (applied to methods or properties)

	Class	Package	Subclass	All
public	x	x	x	x
protected	x	x	x	
default	x	x		
private	x			

  - Optional specifiers
    - Static,abstract,final (synchronized,native,strictfp not for OCA)
- Apply encapsulation principles to a class:
  - Properties should be private (or protected ?)
  - Access to properties should be done by methods: (ex: getters and setters)

- Determine the effect upon object references and primitive values when they are passed into methods that change the values

Handling Exceptions

- Differentiate among checked exceptions, unchecked exceptions, and Errors
    - Class Hierarchy
        - Exception implements Throwable
        - RuntimeException extends Exception
        - Error implements throwable
    - RuntimeException and its subclasses are **unchecked** exceptions
        - It is allowed but not mandatory to handle these exceptions
    - Exception and all its subclasses except those that extends RuntimeException are **checked** exceptions
        - It is mandatory to handle these exception
            - Using a try / catch block
            - Adding a throw <Exception> to the method signature
    - Exception - checked or unchecked - can be thrown by the program or by the JVM
    - Errors are thrown only by the JVM
        - It is possible but discouraged to handle them
- Create a try-catch block and determine how exceptions alter normal program flow
    - If a try block is started there must be a **catch** or a **finally** block
    - The order **try**, **catch** and **finally** block must be respected
    - The braces are mandatory even if there is only one statement
    - The catch blocks are evaluated in the order they have been written, therefore if catching an exception which is a superclass of the following one **will not compile ➜** in this case the catch superclass block would be executed and the subclass catch never.
    - It is possible to catch several exceptions at a time (multi catch)

      ⚠ In this case the exceptions to be caught mustn't be in the same hierarchy
    - ⚠ The finally block is executed even if an exception is thrown in the catch block
    - ⚠ When caught the exceptions are implicitly final (not possible to reassign)
- Describe the advantages of Exception handling
    - Create and invoke a method that throws an exception
    - When overriding a method that does not throw an exception, it is not possible to add **throws** to the method signature.
    - A subclass can declare less exceptions than its superclass
- Recognize common exception classes (such as NullPointerException, ArithmeticException, ArrayIndexOutOfBoundsException, ClassCastException)
    - Types of **runtime (unchecked)** exceptions to be recognized and retained:
        - ArithmeticException: divide by 0
        - ArrayIndexOutOfBoundsException
        - ClassCastException
        - IllegalArgumentException
        - NullPointerException
        - NumberFormatException: wrong string to number conversion

          ⚠ It is thrown programmatically NOT by the JVM
    - Examples of **checked** exceptions:
        - FileNotFoundException

        - IOException

          ⚠ It is thrown programmatically NOT by the JVM
    - Errors: (thrown by the JVM)
        - They are subclasses of **Error** class:
            - ExceptionInInitializerError,StackOverflowError,NoClassDefFoundError
        - They can be handled but it is not advised

Assume the following:

- **Missing package and import statements:** If sample code do not include package or import statements, and the question does not explicitly refer to these missing statements, then assume that all sample code is in the same package, or import statements exist to support them.
- **No file or directory path names for classes:** If a question does not state the file names or directory locations of classes, then assume one of the following, whichever will enable the code to compile and run:
    - All classes are in one file
    - Each class is contained in a separate file, and all files are in one directory
- **Unintended line breaks:** Sample code might have unintended line breaks. If you see a line of code that looks like it has wrapped, and this creates a situation where the wrapping is significant (for example, a quoted String literal has wrapped), assume that the wrapping is an extension of the same line, and the line does not contain a hard carriage return that would cause a compilation failure.
- **Code fragments:** A code fragment is a small section of source code that is presented without its context. Assume that all necessary supporting code exists and that the supporting environment fully supports the correct compilation and execution of the code shown and its omitted environment.
- **Descriptive comments:** Take descriptive comments, such as "setter and getters go here," at face value. Assume that correct code exists, compiles, and runs successfully to create the described effect.