OCA Exam Preparation

JAVA BASICS – REVIEW

ABBREVIATIONS & ACRONYMS

actype – actual object's type at run time
assop – assignment operator
castype – data type specified inside the parens for an explicit cast
comperr – compilation error
ctor – constructor
dim – dimension
initer – initializer
op – operator
paramlist – list of formal parameters in a lambda expression
preditype – data type specified inside the angle brackets
reftype – reference type
refvar – reference variable
sout – any printing statement such as **System.out.println()**, etc.
stat – statement
ternop – ternary operator
var – variable

AIOOBE – ArrayIndexOutOfBoundsException
CCE – ClassCastException
ChE – checked exception
CSR – the Catch-or-Specify Requirement
DTPE – DateTimeParseException
E – an exception (regardless of the type)
IAE – IllegalArgumentException
IOE – IOException
IOOBE – IndexOutOfBoundsException
LDT – any of the new date/time classes in Java 8
LOC – line of code
NFE – NumberFormatException
NPE – NullPointerException
RTE – RuntimeException
SIOOBE – StringIndexOutOfBoundsException
TCF – try-catch-finally construct

igor.soudakevitch@mail.ru

EXAM OBJECTIVE'S STRUCTURE

The 1Z0-808 topics within this group:

1. Define the scope of variables;
2. Define the structure of a Java class;
3. Create executable Java applications with a main method; run a Java program from the command line, including console output;
4. Import other Java packages to make them accessible in your code;
5. Compare and contrast the features and components of Java such as: platform independence, object orientation, encapsulation, etc.

Source: Oracle Univ.

1.1

SCOPE OF VARIABLES

SCOPE OF VARIABLES

- Basically, scope refers to that portion of code where a variable is visible.

- Variables can operate on different levels they can have multiple scopes:
  - class-level variables,
  - instance variables,
  - local variables (including loop vars), and
  - method arguments.

- Local vars are most often defined within the body of a method/ctor and in sub-blocks.

- The scope of a variable is limited by the nearest pair of matching curly braces, {}. For example, the scope of a local variable is less than the scope of a method if the variable was declared within a sub-block inside the method. This sub-block can be:
  - `if` statement
  - `switch` construct
  - any loop
  - TCF construct
  - just a group of assorted stats enclosed by a matching pair of braces.

SCOPE OF VARIABLES, cont'd

- Local variables are not visible (*read*: they cannot be accessed) outside the method or sub-block in which they're defined.
- Instance variables are defined and accessible within an object, which effectively means we need a valid object to work with them. Any instance method of the same class can access these instance variables.
- Class-level variables, a.k.a. `static` variables, are shared by all of the instances of the class; what's more, they can be accessed even if there are no objects of the class.
- Method arguments are used by a method that accepts parameters. Their scope is confined to the method within which they're defined.
- A method argument and a local variable cannot share the same identifier, that is, name.
- Same goes for class and instance variables: they can't be defined using the same name.

SCOPE OF VARIABLES, cont'd

- On the other hand, local and class/instance vars may be defined using the same name. If a method defines a local var that shares the same name with an class/instance variable, the local var shadows the class/instance var (in other words, makes it invisible.)
- Sometimes the variables' scopes overlap with each other; the only way to be sure if a variable is accessible is to identify the block it belongs to (by looking at braces).
- Loop variables are local to the loop within which they're defined.
- All variables go into scope as soon as they get declared.
- Class variables remain in scope as long as the program is running.
- Instance variables go out of scope when the object is garbage collected.
- Local variables go out of scope when the block they are declared in ends.

SCOPE OF VARIABLES, cont'd

```
1  class VarScope{
2      static int x = 4, y;
3      static{
4          x = 44;
5      }
6      int a = 1, b;
7      {
8          b = 11;
9      }
10     void run(int b){
11         int a = b;
12         int c;
13         {
14             // int c = 666;                    // INVALID
15             int x = 444;
16         }
17         for (int d = 0; d < 3; d++){
18             // int a = 3;                      // INVALID
19             int e = 5;
20             e++;
21             System.out.println("e = " + e);    // prints 6 repeatedly
22         }
23     }
24     public static void main(String[] args) {
25         int a = 3;
26         new VarScope().run(a);
27     }
28 }
```

1.2

JAVA CLASS STRUCTURE

JAVA CLASS STRUCTURE

- The structure and functions of Java class are defined in a source code file (**.java** file).
- The compiler creates a single Java bytecode file (**.class** file) for each compiled class, even if it is nested. As for the **.class** files themselves, they are not on the 1Z0-808 exam.
- A class can define multiple components, such as:
  - `package` and `import` declarations,
  - comments,
  - variables,
  - methods,
  - constructors,
  - initialization blocks,
  - *nested classes,*
  - *nested interfaces,*
  - *annotations, and*
  - *enums.*

  these data types are not on the 1Z0-808 exam
- A single .java file can define multiple classes and/or interfaces.
- A public class can be defined only in a source code file with the same name.

JAVA CLASS STRUCTURE, cont'd

- The structure of the source code file directly affects the compilability need to recognize misplaced stats:
  - Java classes are kept inside packages, which group together related classes and interfaces. The packages also provide access protection and namespace management.
  - The `import` stat is used to import **`public`** classes and interfaces from other packages.
  - EXTRA: If a needed `import` statement is missing, classes and interfaces should be referred to by their fully qualified names (in the form of *pack.[subpack.]type_name*).
  - Classes can be imported by class name or wildcard. Wildcards do not make the compiler look inside subpackages (which are mapped to the local file system as subfolders).
  - EXTRA: In the event of a conflict, class name imports take precedence over wildcards.
  - `package` and `import` statements are optional. If present, they apply to all the classes and interfaces defined in the same **.java** file.
  - Fields and methods are also optional but, unlike the package and import stats that must precede the class declaration, they may be placed in any order within the class;
  - If a class defines a `package` statement, it should be the first statement in the **.java** file. It is a comperr if the package stat appears inside or after the class declaration;
  - If a class has a `package` statement, all the imports should follow it.
  - A class can contain exactly one `package` statement + multiple `import` statements;
  - The `import` statement uses simple names of classes and interfaces from within the class.
  - The `import` statement cannot be applied to multiple classes or interfaces with the same simple name;
  - Redundant imports are allowed.
  - EXTRA: The `import` statement requires the dot operator.

JAVA CLASS STRUCTURE, cont'd

- Comments are another component of a class: they are used to annotate Java code and can appear anywhere within the source code file.
- EXTRA: Comments can contain any characters from the entire Unicode charset.
- There are three types of comments:
  - a single-line comment // (a.k.a. 'end-of-line comment'), which hides from the compiler anything that is present to the right-hand side of it;
  - a multiline comment /* */, and
  - EXTRA: a Javadoc comment /** */, which is not on the exam.
- A comment can appear in multiple places, before or after a package statement, before, within, or after the class definition, and before, within, or after the bodies of methods, constructors (ATTN), blocks, loops and so on.
- A Java class may define zero or more members such as static fields, instance variables, methods, or constructors whose definitions can be placed in any order within the class.

  Corollary: Since the declaration order does not matter, a method may use, for example, an instance variable even before it has been declared in the file.

JAVA CLASS STRUCTURE, cont'd

```
1  /* File: Test.java
2   * This is a simple illustration of the rules
3   * that concern Java class structure
4  */
5
6  package org.xlator;
7  import java.lang.*;
8  // package org.xlator;           // INVALID
9  // import java.util.*;           // VALID but the code uses another approach
10 import java.util.ArrayList;
11 import java.util.Date;
12 // import java.sql.Date;         // INVALID
13
14 interface I1{}
15 // public interface I2{}        // INVALID
16
17 class C1{ }
18 public class Test {
19     public static void main(String[] args) {
20         System.out.println(new Test().list.add("Hello"));    // prints true
21     }
22     java.util.List<String> list = new ArrayList<String>();
23 }
```

1.3

`main()` method, etc.

`main()` method, cont'd

- Before any Java class can be used by the JVM, it must be compiled into bytecode:

  `javac MyClass.java` → creates the bytecode file **MyClass.class**

- Java classes can be either executable or non-executable. An executable Java class runs when its class name (together with optional parameters) is handed over to the JVM:

  `java MyClass 1 2 3` → no extension! it's our class name, not the filename!

- To be executable, the Java class must define the **main()** method, a.k.a. 'entry point', at which the JVM begins program execution.

- The most commonly used signature of the **main()** method is:

```
public static void main(String[] args)
```

- EXTRA: also acceptable are the varargs:

```
public static void main(String... args)
```

`main()` method, cont'd

- To make the class executable, its **main()** method must be both public and static.
- A class can define multiple methods with the name **main()**, provided that their signatures do not match the signature of the **main()** method defined as the program's entry point.
- The 'entry-point' **main()** accepts an array of type **String** that will contain the parameters specified on the command line after the class name. In fact, the **main()** method uses copies of these parameters as its arguments.
- Arguments are referenced starting with **args[0]**. An attempt to access an argument that wasn't passed in causes an RTE, namely, AIOOBE.
- EXTRA: **String**[] **args** in public static void main() is never null. If the program is run without any command line arguments, **args** points to a **String** array of zero length.

1.4

IMPORTING PACKAGES

IMPORTING PACKAGES

- Java code is commonly organized into packages, which resemble file system folders.
- To reference public classes or interfaces contained in other packages, we can use:
  - either an `import` statement, or
  - the fully qualified name of the class/interface.
- No `import` statement can be placed before a `package` declaration.
- An `import` statement can end with either simple name of the class or a wildcard symbol, *.
- If an `import` statement ends with the asterisk, it means that the code wants to have access to all `public` data types in that particular package.
- On the other hand, the wildcard symbol does not provide access to any of the subpackages that may exist inside that one.
- The **java.lang** package is the only package the compiler imports automatically.
- Packages can contain multiple subpackages depending on how programmers organize their classes and interfaces.

IMPORTING PACKAGES, cont'd

- By default, data types in different packages and subpackages aren't visible to each other.
- On the other hand, all classes and interfaces within the same package are visible to each other.
- The package and subpackage names are chained together with the dot operator.
- An import statement is nothing but a handy way to tell the compiler where to look for a specific public class or interface;

  Corollary: Instead of using an import stat, we can always write the fully qualified name of the public class or interface right in our code – but it soon becomes tedious.

- EXTRA: It also follows that import statements can't be used to access multiple classes or interfaces that share the same simple name but reside in different packages. This problem can be solved, for example, by using an import stat for one conflicting class and the fully qualified name for its competitor. Another approach is to re-think the entire architecture of your namespace.

IMPORTING PACKAGES, cont'd

- An `import` stat allows to gain access to either a single `public` member of the package, or to all of them (by using the above-mentioned wildcard symbol).

- **Known trap on the exam:** If a data type isn't `public`, it cannot be imported:

```
package pack1;
class A {}

package pack2;
import pack1.*;
// import pack1.A;                  // INVALID: "A is not public in pack1; cannot
                                       be accessed from outside package"
// class B extends A {}             // INVALID: ditto
// class C extends pack1.A {}       // INVALID: ditto
```

- Unlike the asterisk used to access file system subfolders, `import` stat's wildcard symbol does not work in a similar fashion with subpackages. In other words, * does not import subpackages.

IMPORTING PACKAGES, cont'd

- EXTRA: If the source code file does not contain a `package` declaration, the class / interface is considered to be a member of the so-called default package.
- EXTRA: The default package's members are accessible only to the classes or interfaces defined in the same directory that contains the default package.
- EXTRA: A class belonging to a default package can't be used in any named packaged class regardless of whether it is defined within the same directory, or not.
- It is possible to import an individual `public static` member of a class or all of its `public static` members by using an import static declaration.
- To import a `static` method, we must use its name only, that is, without parentheses.
- If the code appears clean and the LOCs are numbered, be on guard for missing imports for:
  - lists (**List** and **ArrayList**)
  - predicative lambdas
  - LDT-related LOCs
  - **Arrays** (e.g., **Arrays.asList()**)
  - **Collections** (e.g., **Collections.sort()**)

1.5

JAVA OOP FEATURES

JAVA OOP FEATURES

- Java is a computer programming language that is concurrent, class-based and object-oriented. The advantages of object-oriented software development:
  - Modular development of code, which improves its robustness;
  - Reusability of code;
  - Code is more flexible and dynamic at run time;
  - Better maintainability of code.
- The OOP-based development is supported by a number of built-in features, such as encapsulation, inheritance, polymorphism, and abstraction.
- Encapsulation allows objects to hide their internal characteristics and implementation details. Each object can provide a number of methods, which are accessible to other objects thus permitting them to read and/or change its internal state.
- Encapsulation is realized through the use of access modifiers.

JAVA OOP FEATURES, cont'd

- Some of the advantages of encapsulation:
  - The internal state of every object can be protected;
  - Encapsulation improves usability and maintainability of code because the behavior of an object can be modified independently of other data types;
  - Encapsulation decreases coupling, that is, interaction between and among classes thus improving modularity of the design.
- Polymorphism is the ability to realize behavior depending on the underlying data type. A polymorphic data type is the type whose methods can also be applied to variables of some other related type. In other words, we can use the same variable to refer to different types and thus a method call can perform different tasks depending on the type of the actual object.
- Polymorphism makes the code more dynamic at run time and also improves its flexibility and reusability.
- Inheritance provides an object with the ability to acquire the fields and methods of its parent class, also called superclass or base class. Inheritance further enhances reusability of code by adding new features to an existing data type without modifying it.

JAVA OOP FEATURES, cont'd

- Java does not support multiple inheritance. Each class is permitted to extend only one class, but can implement multiple interfaces.
- EXTRA: Abstraction allows to develop data types in terms of their own functionality instead of implementation details. Java provides support for abstract classes that expose interfaces to the outside world without including the actual implementation of all methods. Basically, abstraction aims to separate the implementation details of a class from its behavior.
- EXTRA: Difference between abstraction and encapsulation:
  - Abstraction and encapsulation are complementary concepts: while abstraction deals with the behavior of an object, encapsulation concerns the implementation details of this behavior.
  - Encapsulation is usually achieved by hiding information about the internal state of an object and thus can be seen as a way to realize abstraction.
- Java is platform independent, meaning that it allows to develop applications that can run, without having to be rewritten or recompiled, on any platform that has Java Runtime Environment.

EXAM OBJECTIVE 1, WARMING-UP EXERCISES

TIME TO EXERCISE

TEST