C3 Assignments

Java 8 Associate

1Z0-808

Link

https://mylearn.oracle.com/ou/exam/java-se-8-programmer-i-1z0-808/105037/110679/170387
https://docs.oracle.com/javase/specs/jls/se8/html
https://docs.oracle.com/javase/tutorial
http://www.java2s.com

https://enthuware.com
https://github.com/carlosherrerah/OCA

JUAN CARLOS HERRERA HERNANDEZ
carlos.herrera@upa.edu.mx

Contenido

>>

1. Stack and Heap—Quick Review

Understanding the basics of the stack and the heap makes it far easier to understand topics like argument passing, polymorphism, threads, exceptions, and garbage collection. The various pieces (methods, variables, and objects) of Java programs live in one of two places in memory: the stack or the heap.

- Instance variables and objects live on the heap.
- Local variables (method variables) live on the stack.

Scope: "How long will this variable live?"
Remember the four basic scopes in order of lessening life span: static, instance, local, and block.

Local variables MUST be assigned a value explicitly.

The heap is where objects live and where all the cool garbage collection activity takes place.

You (the programmer) can request a garbage collection run, but you can't force it.
`finalize()` method.

>>

2. Literals, Assignments, and Variables

- Integer literals can be binary (0b10), decimal, octal (such as `013`), or hexadecimal (such as `0x3d`).
- Literals for `long`s end in `L` or `l`. (For the sake of readability, we recommend "`L`".)
- Float literals end in `F` or `f`, and `double` literals end in a digit or `D` or `d`. (default)
- The `boolean` literals are `true` and `false`.
- Literals for `char`s are a single character inside single quotes: `'d'`.

There are four ways to represent integer numbers in the Java language: Binario, Octal, Decimal, Hexadecimal.

`int b = 0b1010; //10`
`int B = 0B1010;`
	`int o = 015;`
	`int O = 015;`
		`int d = 10;`	`int h = 0xFea;`
			`int H = 0XFE0;`

Numeric Literals with Underscores:

- you CANNOT use the underscore literal at the beginning or end of the literal.
- you CANNOT add an underscore character directly next to the decimal point.
- you CANNOT add an underscore character directly next to the X or B in hex or binary numbers

A variable of type float (32 bits), you must attach the suffix F or f to the number.
A boolean value can be defined only as true or false.
Characters are just 16-bit unsigned integers (0 to 65535).

`int a = 1000000;`
`int a = 1_000_000;`
`int a = 1_000___000;`
`int a = 1_00_0000;`
`int a = 0b1111_0000;`
`long a = 1000;`
`long a = 1000L;`
 	
`float a = 1_0.001_002f;`
`double a = 3.1416;`
`double a = 3.1416d;`
`double a = 314.16e-2;`
`boolean a = true;`
	`char A = 'A';`
	`char _A = 65;`
	`char a = 'a';`
	`char _a = 97;`
	`char unicode_A = '\u0041';`
	`char unicode_a = '\u0061';`
	
	`char nullChar = '\u0000';`
	`char nextLine = '\n';`
	`char doubleQuote = '\"';`
	
	`String curso = "JAVA";`
		`Illegal`
		`// int a = _1000_000;`
		`// int a = 0b_1111_0000;`
		`// int a = 0x_1_0;`
		`// int a = 0x1_0_;`
		`// float a = 1_0._001_002f;`
		`// boolean a = TRUE;`

Add two bytes together and you'll get an int.
Multiply an int and a short and you'll get an int.
Divide a short by a byte and you'll get…an int

byte c = (byte) (128); // -128
`long a = 130L`
`byte b = (byte) a;`
	`float pi = 3.1416F;`
	`int entera = (int) pi`
		`byte b = 3;`
		`b+=7;`
		`b = b + 7 // Error`
		`b = (byte) ( b +7 );`

```
Animal a = new Animal();
```

The preceding line does three key things:

- Makes a reference variable named `a`, of type `Animal`
- Creates a new `Animal` object on the heap
- Assigns the newly created `Animal` object to the reference variable `a`

```
Animal a = null;  // Candidata a Garbage Collection
```

A wrapper object is an object that holds the value of a primitive.
Every kind of primitive has an associated wrapper class: `Boolean`, `Byte`, `Character`, `Double`, `Float`, `Integer`, `Long`, and `Short`.

```
Boolean B3 = new Boolean("TrUe");
```

>>

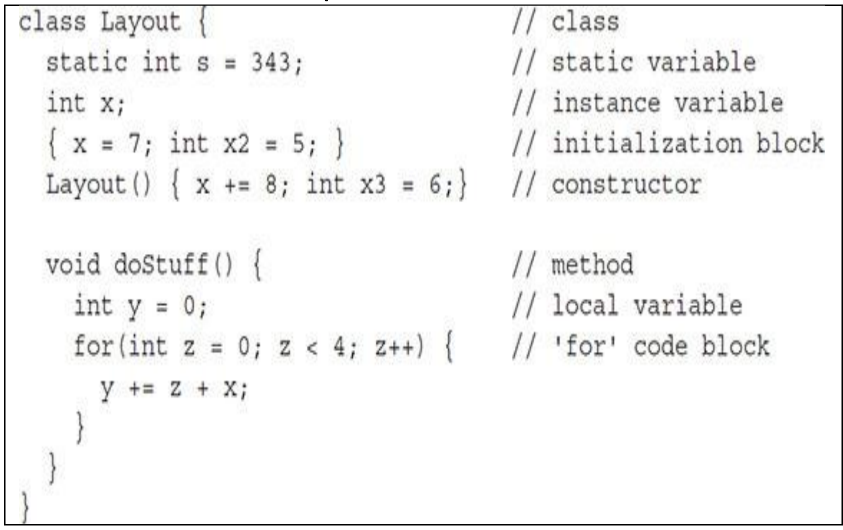
3. Scope (OCA Objective 1.1)

Determine the scope of variables.

Scope refers to the lifetime of a variable.
There are four basic scopes:

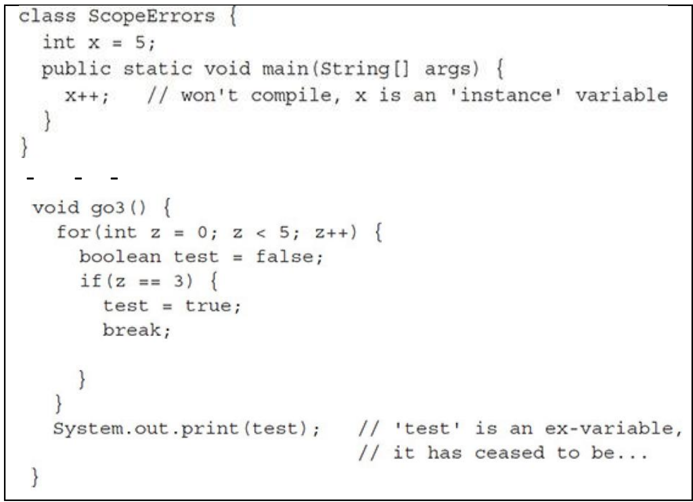
- Static variables live basically as long as their class lives.
- Instance variables live as long as their object lives.
- Local variables live as long as their method is on the stack; however, if their method invokes another method, they are temporarily unavailable.
- Block variables (for example, in a `for` or an `if`) live until the block completes.

Pay extra attention to code-block scoping errors. You might see them in switches, try-catches, for, do, and while loops.

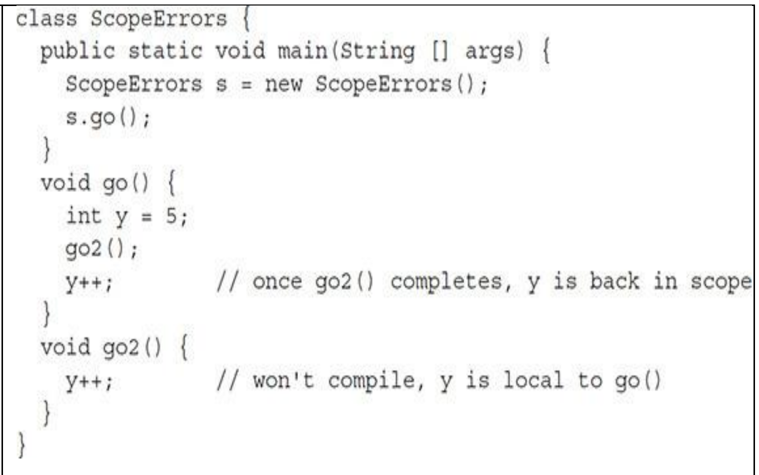


```
class Layout {                          // class
  static int s = 343;                   // static variable
  int x;                                // instance variable
  { x = 7; int x2 = 5; }                // initialization block
  Layout() { x += 8; int x3 = 6;}       // constructor

  void doStuff() {                      // method
    int y = 0;                          // local variable
    for(int z = 0; z < 4; z++) {        // 'for' code block
      y += z + x;
    }
  }
}
```

- `s` is a static variable.
- `x` is an instance variable.
- `y` is a local variable (sometimes called a “method local” variable).
- `z` is a block variable.
- `x2` is an `init` block variable, a flavor of local variable.
- `x3` is a constructor variable, a flavor of local variable.



```
class ScopeErrors {
  int x = 5;
  public static void main(String[] args) {
    x++;   // won't compile, x is an 'instance' variable
  }
}
 -   -  -
 void go3() {
   for(int z = 0; z < 5; z++) {
     boolean test = false;
     if(z == 3) {
       test = true;
       break;

     }
   }
   System.out.print(test);   // 'test' is an ex-variable,
                             // it has ceased to be...
 }
```



```
class ScopeErrors {
  public static void main(String [] args) {
    ScopeErrors s = new ScopeErrors();
    s.go();
  }
  void go() {
    int y = 5;
    go2();
    y++;        // once go2() completes, y is back in scope
  }
  void go2() {
    y++;        // won't compile, y is local to go()
  }
}
```

>>

4. Variable Initialization

2.1 Declare and initialize variables (including casting of primitive datatypes).
4.1 Declare, instantiate, initialize and use a one-dimensional array
4.2 Declare, instantiate, initialize and use multi-dimensional array

4.1. Basic Assignments

- Literal integers are implicitly `int`s.
- Integer expressions always result in an `int`-sized result, never smaller.
- Floating-point numbers are implicitly doubles (64 bits).
- Narrowing a primitive truncates the *high order* bits.
- Compound assignments (such as `+=`) perform an automatic cast.
- A reference variable holds the bits that are used to refer to an object.
- Reference variables can refer to subclasses of the declared type but not to superclasses.
- When you create a new object, such as `Button b = new Button();`, the JVM does three things:
  - Makes a reference variable named `b`, of type `Button`.
  - Creates a new `Button` object.
  - Assigns the `Button` object to the reference variable `b`.

4.2. Using a Variable or Array Element That Is Uninitialized and Unassigned

- When an array of objects is instantiated, objects within the array are not instantiated automatically, but all the references get the default value of `null`.
- When an array of primitives is instantiated, elements get default values.
- Instance variables are always initialized with a default value.
- Local/automatic/method variables are never given a default value. If you attempt to use one before initializing it, you'll get a compiler error.

Default Values for Primitives and Reference Types

Variable Type	Default Value
Object reference	`null` (not referencing any object)
`byte`, `short`, `int`, `long`	`0`
`float`, `double`	`0.0`
`boolean`	`false`
`char`	`'\u0000'`

It's a good idea to initialize all your variables, even if you're assigning them with the default value. Your code will be easier to read;

Array elements are given their default values (`0`, `false`, `null`, `'\u0000'`, and so on) regardless of whether the array is declared as an instance or local variable.

int[] a = new int[5]; // Es un array de 5 elementos, todos inicializados a 0

When using the word **`new`** it is therefore an object.

The behavior of Strings is extremely important in the exam, so we'll cover it in much more detail in Chapter 6.

`String` objects are immutable; you can't change the value of a `String` object.

```
class StringTest {
  public static void main(String [] args) {
    String x = "Java";  // Assign a value to x
    String y = x;       // Now y and x refer to the same
                        // String object

    System.out.println("y string = " + y);
    x = x + " Bean";    // Now modify the object using
                        // the x reference
    System.out.println("y string = " + y);
  }
}
```

```
1. String s = "Fred";
2. String t = s;      // Now t and s refer to the same
                      // String object
3. t.toUpperCase();   // Invoke a String method that changes
                      // the String
```

When line 2 completes, both t and s reference the same String object. But when line 3 runs, rather than modifying the object referred to by t and s (which is the one and only String object up to this point), a brand new String object is created. And then it's abandoned. Because the new String isn't assigned to a String variable, the newly created String (which holds the string "FRED") is toast.

>>

5. Passing Variables into Methods

- Methods can take primitives and/or object references as arguments.
- Method arguments are always copies.
- Method arguments are never actual objects (they can be references to objects).
- A primitive argument is an unattached copy of the original primitive.
- A reference argument is another copy of a reference to the original object.
- Shadowing occurs when two variables with different scopes share the same name. This leads to hard-to-find bugs and hard-to-answer exam questions.

Java siempre pasa los argumentos por valor, pero el comportamiento puede parecer diferente dependiendo de si se trata de un tipo primitivo o un objeto.

Shadowing involves reusing a variable name that's already been declared somewhere else.

>>

6. Garbage Collection

Explain an object's lifecycle (creation, "dereference by reassignment," and garbage collection)

- In Java, garbage collection (GC) provides automated memory management.
- The purpose of GC is to delete objects that can't be reached.
- Only the JVM decides when to run the GC; you can only suggest it.
- You can't know the GC algorithm for sure.
- Objects must be considered eligible before they can be garbage collected.
- An object is eligible when no live thread can reach it.
- To reach an object, you must have a live, reachable reference to that object.
- Java applications can run out of memory.
- Islands of objects can be garbage collected, even though they refer to each other.
- Request garbage collection with `System.gc();`.
- The `Object` class has a `finalize()` method.
- The `finalize()` method is guaranteed to run once and only once before the garbage collector deletes an object.
- The garbage collector makes no guarantees; `finalize()` may never run.
- You can ineligible-ize an object for GC from within `finalize()`.

The heap is that part of memory where Java objects live, and it's the one and only part of memory that is in any way involved in the garbage collection process. So all garbage collection revolves around making sure the heap has as much free space as possible.

For the purpose of the exam, what this boils down to is deleting any objects that are no longer reachable by the Java program running. The important concept for you to understand for the exam is: When does an object become eligible for garbage collection?

When Does the Garbage Collector Run?

The garbage collector is under the control of the JVM; the JVM decides when to run the garbage collector. From within your Java program, you can ask the JVM to run the garbage collector; but there are no guarantees, under any circumstances, that the JVM will comply. the JVM will decide to ignore your request.

Every Java program has from one to many threads. All you need to know is that threads can be alive or dead.

Nulling a Reference

An object becomes eligible for garbage collection when there are no more reachable references to it. The first way to remove a reference to an object is to set the reference variable that refers to the object to null.

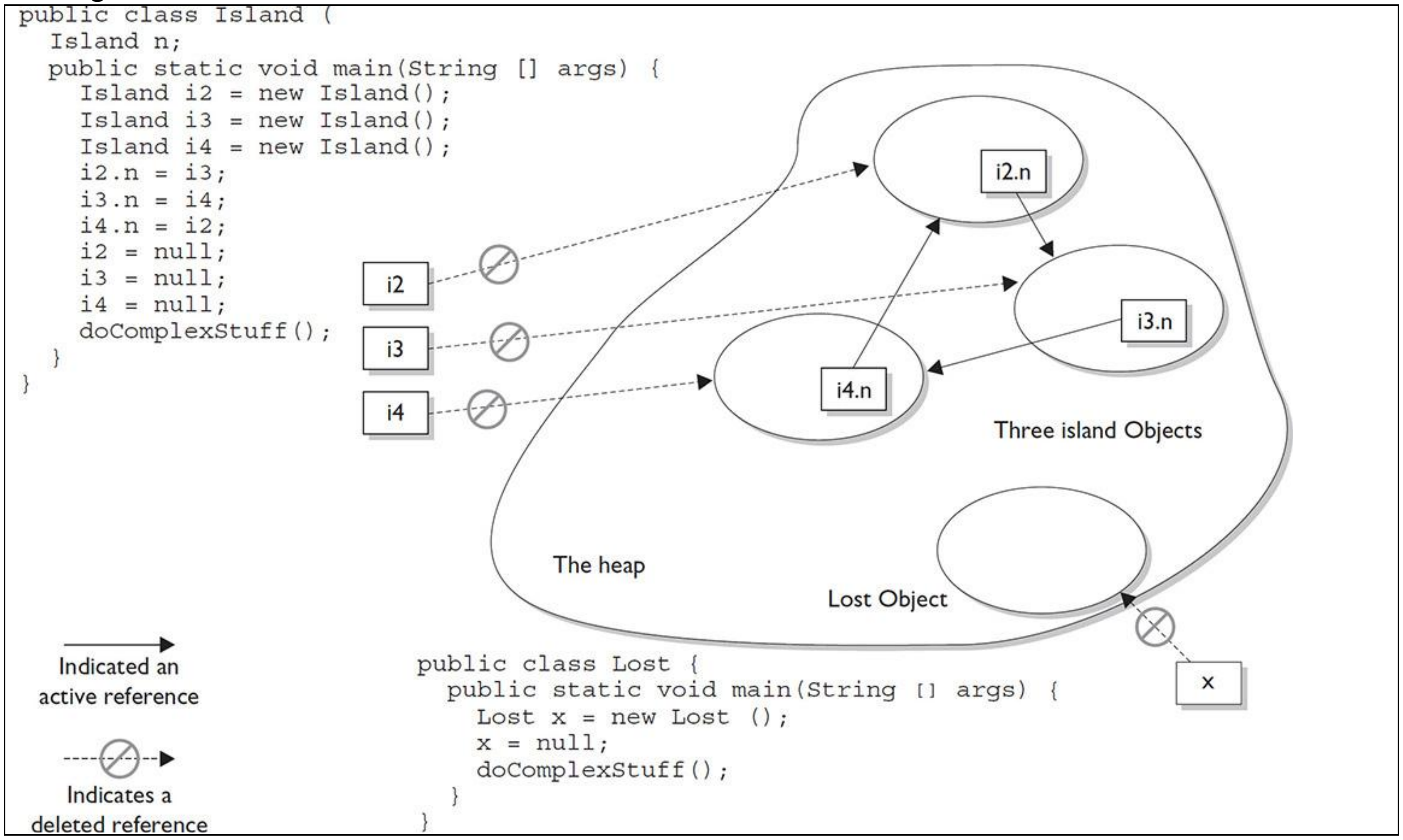
StringBuffer sb = new StringBuffer("hello"); // `sb` is not eligible for collection
sb = null; // Now the StringBuffer object is eligible for collection

```
class GarbageTruck {
      public static void main(String [] args) {
            StringBuffer s1 = new StringBuffer("hello");
            StringBuffer s2 = new StringBuffer("goodbye");
            System.out.println(s1);
            // At this point the StringBuffer "hello" is not eligible
            s1 = s2; // Redirects s1 to refer to the "goodbye" object
            // Now the StringBuffer "hello" is eligible for collection
      }
}
```

Objects that are created in a method also need to be considered. When a method is invoked, any local variables created exist only for the duration of the method. Once the method has returned, the objects created in the method are eligible for garbage collection. There is an obvious exception, however. If an object is returned from the method, its reference might be assigned to a reference variable in the method that called it; hence, it will not be eligible for collection. Examine the following code:

```
package scjp.c3;
import java.util.Date;

public class P6GarbageC {

  public static void doComplicatedStuff() {}

  public static void main(String[] args) {
    Date d = getDate();
    doComplicatedStuff();
    System.out.println("d = " + d);
  }

  public static Date getDate() {
    Date d2 = new Date();
    StringBuffer now = new StringBuffer(d2.toString());
    System.out.println(now); // candidate for garbage collection
    return d2;               // Not a candidate for garbage collection
  }

}
```

Isolating a Reference



```
public class Island (
  Island n;
  public static void main(String [] args) {
    Island i2 = new Island();
    Island i3 = new Island();
    Island i4 = new Island();
    i2.n = i3;
    i3.n = i4;
    i4.n = i2;
    i2 = null;
    i3 = null;
    i4 = null;
    doComplexStuff();
  }
}
```

```
public class Lost {
  public static void main(String [] args) {
    Lost x = new Lost ();
    x = null;
    doComplexStuff();
  }
}
```

>>