



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/carlosherrera/OCA>

JUAN CARLOS HERRERA HERNANDEZ

carlos.herrera@upa.edu.mx

Contenido

1. Stack and Heap—Quick Review	3
2. Literals, Assignments, and Variables	4
3. Scope (OCA Objective 1.1)	6
4. Variable Initialization.....	7
4.1. Basic Assignments	7
4.2. Using a Variable or Array Element That Is Uninitialized and Unassigned	7
5. Passing Variables into Methods \$286	9
6. Garbage Collection	10
7. Java Operators.....	11
8. Using if and switch Statements	11
9. Creating Loops Constructs	11
10. Handling Exceptions	11
11. Common Exceptions and Errors	11

>>

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 `longs` 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 `chars` are a single character inside single quotes: `'d'`.

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

<code>int b = 0b1010; //10</code>	<code>int o = 015;</code>	<code>int d = 10;</code>	<code>int h = 0xFEa;</code>
<code>int B = 0B1010;</code>	<code>int O = 015;</code>		<code>int H = 0xFE0;</code>

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).

<pre>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;</pre>	<pre>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";</pre>	<pre>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;</pre>
---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	----------------------------------------------------------------------------------------------------------------------------------------------------------

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

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

```
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.

<pre> class Layout { static int s = 343; int x; { x = 7; int x2 = 5; } Layout() { x += 8; int x3 = 6;} void doStuff() { int y = 0; for(int z = 0; z < 4; z++) { y += z + x; } } } </pre>	<ul style="list-style-type: none"> • <code>s</code> is a static variable. • <code>x</code> is an instance variable. • <code>y</code> is a local variable (sometimes called a “method local” variable). • <code>z</code> is a block variable. • <code>x2</code> is an <code>init</code> block variable, a flavor of local variable. • <code>x3</code> is a constructor variable, a flavor of local variable.
----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

<pre> 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... } </pre>	<pre> 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() } } </pre>
---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

>>

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 `ints`.
- 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 \$\$286

- 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.

>>

6. 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()`.

>>

7. Java Operators

8. Using if and switch Statements

9. Creating Loops Constructs

10. Handling Exceptions

11. Common Exceptions and Errors