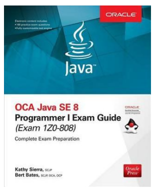

ORACLE

Java

OCA Java SE 8 Programmer I Exam Guide (Exam 1Z0-808)

Complete Exam Preparation

Kathy Sierra, Bert Bates

Chapter 3

Assignments

Business Training

Content

- **Stack and Heap—Quick Review**
- **Literals, Assignments, and Variables**
  (OCA Objectives 2.1, 2.2, and 2.3)
- **Scope**
  (OCA Objective 1.1)
- **Variable Initialization**
  (OCA Objectives 2.1, 4.1, and 4.2)
- **Passing Variables into Methods**
  (OCA Objective 6.6)
- **Garbage Collection**
  (OCA Objective 2.4)

Stack and Heap—Quick Review

The Stack and the Heap

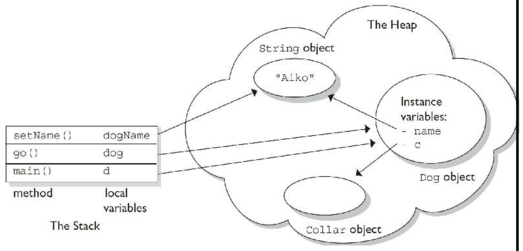
- The various pieces (methods, variables, and objects) of Java programs live in one of two places in memory: the stack or the heap.
  - Local variables live on the stack.
  - Instance variables and objects live on the heap.

```
class Collar { }

class Dog {
  Collar c;                        // instance vari
  String name;                     // instance vari

  public static void main(String [] args) {

    Dog d;                         // local variabl
    d = new Dog();
    d.go(d);
  }
  void go(Dog dog) {               // local variabl
    c = new Collar();
    dog.setName("Aiko");
  }
  void setName(String dogName) {   // local var: do
    name = dogName;
    // do more stuff
  }
}
```



Literals, Assignments, and Variables
(OCA Objectives 2.1, 2.2, and 2.3)

Literal Values for All Primitive Types

- **Integer Literals**
  - There are three ways to represent integer numbers in the Java language: decimal (base 10), octal (base 8), and hexadecimal (base 16).
  - Most exam questions with integer literals use decimal representations, but the few that use octal or hexadecimal are worth studying for.
    - **Decimal Literals** They are represented as is, with no prefix of any kind, as follows:

      ```
      int length = 343;
      ```

    - **Octal Literals** Octal integers use only the digits 0 to 7. In Java, you represent an integer in octal form by placing a zero in front of the number, as follows:

      ```
      int nine = 011;   // Equal to decimal 9
      ```

    - **Hexadecimal Literals Hexadecimal** You are allowed up to 16 digits in a hexadecimal number, not including the prefix Ox or the optional suffix extension L. All of the following hexadecimal assignments are legal:

      ```
      int x = 0X0001; int y = 0x7fffffff; int z = 0xDeadCafe; long jo = 110599L;
      ```

- **Floating-Point Literals**
  - Floating-point literals are defined as double (64 bits) by default, so if you want to assign a floating-point literal to a variable of type float (32 bits), you must attach the suffix F or f to the number.

    ```
    float g = 49837849.029847F;
    ```

Literal Values for All Primitive Types

- **Boolean Literals**
  - A boolean value can only be defined as true or false.

```
boolean t = true;  // Legal
boolean  f = 0;    // Compiler error!
```

- **Character Literals**
  - A char literal is represented by a single character in single quotes.

```
char a = 'a';
char letterN = '\u004E'; // The letter 'N' in Unicode
char a = 0x892;        // hexadecimal literal
char d = (char) -98;    // Ridiculous, but legal
```

Assignment Operators

- **Primitive Assignments**
  - The equal (=) sign is used for assigning a value to a variable, and it's cleverly named the assignment operator.

```
int x = 7;     // literal assignment
```

- **Primitive Casting**
  - Casting lets you convert primitive values from one type to another.
  - First we'll look at an implicit cast:

```
int a = 100;
long b = a; // Implicit cast, an int value always fits in a long
```

  - An explicit casts looks like this:

```
float a = 100.001f;
int b = (int)a; // Explicit cast, the float could lose info
```

- **Variable Scope**
  - For the purposes of discussing the scope of variables, we can say that there are four basic scopes:
    1. Static variables have the longest scope; they are created when the class is loaded, and they survive as long as the class stays loaded in the JVM.
    2. Instance variables are the next most long-lived; they are created when a new instance is created, and they live until the instance is removed.
    3. Local variables are next; they live as long as their method remains on the stack. As we'll soon see, however, local variables can be alive, and still be "out of scope".
    4. Block variables live only as long as the code block is executing.

Scope
(OCA Objective 1.1)

Variable scope

- There are four basic scopes:
  1. **Static variables have the longest scope**; they are created when the class is loaded, and they survive as long as the class stays loaded in the Java Virtual Machine (JVM).
  2. **Instance variables are the next most long-lived**; they are created when a new instance is created, and they live until the instance is removed.
  3. **Local variables are next**; they live as long as their method remains on the stack. As we'll soon see, however, local variables can be alive and still be "out of scope."
  4. **Block variables live only as long as the code block is executing**.

- Attempting to access an instance variable from a static context (typically from main()):

```
class ScopeErrors {
  int x = 5;
  public static void main(String[] args) {
    x++;    // won't compile, x is an 'instance' variable
  }
}
```

Variable scope

- Attempting to access a local variable of the method that invoked you. When a method, say **go()**, invokes another method, say **go2()**, **go2()** won't have access to **go()**'s local variables. While **go2()** is executing, **go()**'s local variables are still *alive*, but they are *out of scope*.

```
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()
  }
}
```

- Attempting to use a block variable after the code block has completed. It's very common to declare and use a variable within a code block, but be careful not to try to use the variable once the block has completed:

```
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...
}
```

Variable Initialization

(OCA Objectives 2.1, 4.1, and 4.2)

Using a Variable or Array Element that is Uninitialized and Unassigned

- **Local variables are sometimes called stack, temporary, automatic, or method variables, but the rules for these variables are the same regardless of what you call them. Although you can leave a local variable uninitialized, the compiler complains if you try to use a local variable before initializing it with a value, as we shall see.**
- **Primitive and Object Type Instance Variables**
  - Instance variables (also called member variables) are variables defined at the class level.
  - 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'`

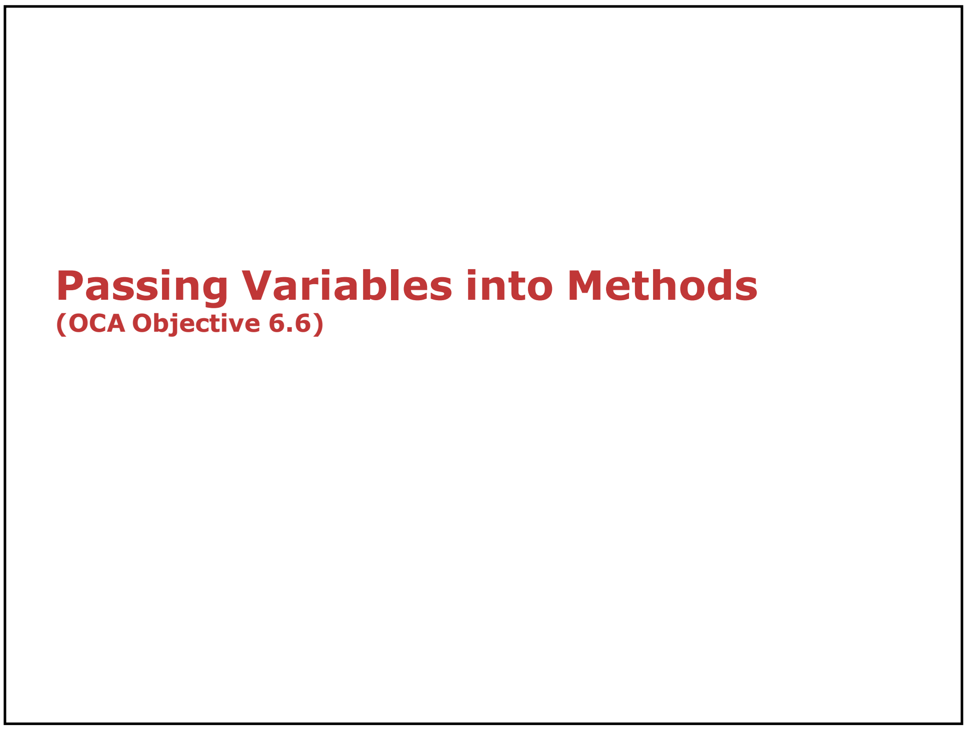
- **Local (Stack, Automatic) Primitives and Objects**
  - Local variables, including primitives, always, always, always must be initialized before you attempt to use them (though not necessarily on the same line of code). Java does not give local variables a default value
- **Local Object References**
  - Objects references, too, behave differently when declared within a method rather than as instance variables. With instance variable object references, you can get away with leaving an object reference uninitialized, as long as the code checks to make sure the reference isn't null before using it.

String Class is immuable !

- **One exception to the way object references are assigned is String. In Java, String objects are given special treatment. For one thing, String objects are immutable; you can't change the value of a String object :**

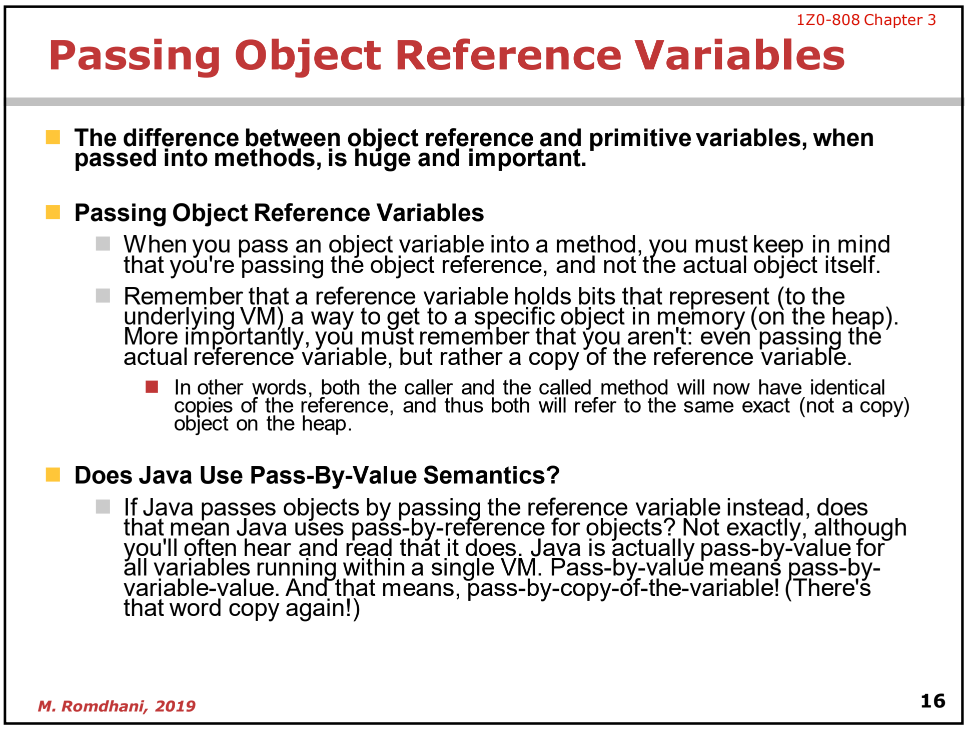
```
class StringTest {
  public static void main(String [] args) {
    String x = "Java";    // Assign a value to x
    String y = x;         // Now y and x refer to the same String object
    System.out.println("y string = " + y);
    x = x + " Bean";      // Now modify the object using the x reference
    System.out.println("y string = " + y);
  }
}
%java StringTest
y string = Java
y string = Java
```

- **You need to understand what happens when you use a String reference variable to modify a string:**
  - A new string is created (or a matching String is found in the String pool), leaving the original String object untouched.
  - The reference used to modify the String (or rather, make a new String by modifying a copy of the original) is then assigned the brand new String object.



Passing Variables into Methods

(OCA Objective 6.6)



1Z0-808 Chapter 3

Passing Object Reference Variables

- **The difference between object reference and primitive variables, when passed into methods, is huge and important.**
- **Passing Object Reference Variables**
  - When you pass an object variable into a method, you must keep in mind that you're passing the object reference, and not the actual object itself.
  - Remember that a reference variable holds bits that represent (to the underlying VM) a way to get to a specific object in memory (on the heap). More importantly, you must remember that you aren't: even passing the actual reference variable, but rather a copy of the reference variable.
    - In other words, both the caller and the called method will now have identical copies of the reference, and thus both will refer to the same exact (not a copy) object on the heap.
- **Does Java Use Pass-By-Value Semantics?**
  - If Java passes objects by passing the reference variable instead, does that mean Java uses pass-by-reference for objects? Not exactly, although you'll often hear and read that it does. Java is actually pass-by-value for all variables running within a single VM. Pass-by-value means pass-by-variable-value. And that means, pass-by-copy-of-the-variable! (There's that word copy again!)

M. Romdhani, 2019 16

Passing Primitive Variables

- **When a primitive variable is passed to a method, it is passed by value, which means pass-by-copy-of-the-bits-in-the-variable.**

```
class ReferenceTest {
 public static void main (String [] args) {
  int a = 1;
  ReferenceTest rt = new ReferenceTest();
  System.out.println("Before modify() a = " + a);
  rt.modify(a);
  System.out.println("After modify() a = " + a) ;
 }
 void modify(int number) {
  number = number + 1;
  System.out.println("number = " + number);
 }
}
```

  - The resulting output looks like this:

```
Before modify() a = 1
number = 2
After modify() a = 1
```

Garbage Collection
(OCA Objective 2.4)

Overview of Memory Management and Garbage Collection

- **Java's garbage collector provides an automatic solution to memory management.**
  - In most cases it frees you from having to add any memory management logic to your application.
  - The downside to automatic garbage collection is that you can't completely control when it runs and when it doesn't.
- **Overview of Java's Garbage Collector**
  - 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.
- **When Does the Garbage Collector Run?**
  - 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.
- **How Does the Garbage Collector Work?**
  - You just can't be sure. The important concept to understand for the exam is when does an object become eligible for garbage collection?

Writing Code that Explicitly Makes Objects Eligible for Collection

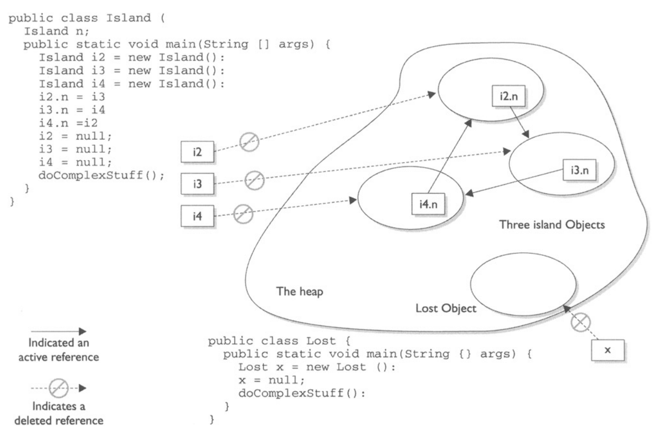
- **Nulling a Reference**
  - As we discussed earlier, an object becomes eligible for garbage collection when there are no more reachable references to it.
- **Reassigning a Reference Variable**

```
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
 }
}
```

  - Exception :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.

Writing Code that Explicitly Makes Objects Eligible for Collection

- **Isolating a Reference**
  - There is another way in which objects can become eligible for garbage collection, even if they still have valid references! We call this scenario "islands of isolation."



```
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():
  }
}
```

Writing Code that Explicitly Makes Objects Eligible for Collection

- **Forcing Garbage Collection**
  - In reality, it is possible only to suggest to the JVM that it perform garbage collection. However, there are no guarantees the JVM will actually remove all of the unused objects from memory (even if garbage collection is run). It is essential that you understand this concept for the exam.
  - The simplest way to ask for garbage collection (remember—just a request) is
    - System.gc();
- **Cleaning up before Garbage Collection—The finalize() Method**
- **Java provides you a mechanism to run some code just before your object is deleted by the garbage collector.**
  - This code is located in a method named finalize() that all classes inherit from class Object.
- **Tricky Little finalize() Gotcha's**
  - There are a couple of concepts concerning finalize() that you need to remember.
    - For any given object, finalize () will be called only once (at most) by the garbage collector.
    - Calling finalize() can actually result in saving an object from deletion.