
Core Java 8
Assignments
Capgemini

Lesson Objectives

After completing this lesson, participants will be able to

- Understand the different types of Arrays
- Implement one and multi dimensional arrays
- Iterate arrays using loops
- Garbage Collection
- Memory Management

Stack and Heap – Overview

Stack is used for static memory allocation and Heap is used for dynamic memory allocation

Variables allocated on the stack are stored directly to the memory and access to this memory is very fast, and it's allocation is dealt with when the program is compiled.

Variables allocated on the heap have their memory allocated at run time and accessing this memory is a bit slower, but the heap size is only limited by the size of virtual memory.

Literals, Assignments, and Variables

Literals represents value to be assigned for variable.

Java has three types of literals:

- Primitive type literals
- String literals
- null literal

Primitive literals are further divided into four subtypes:

- Integer
- Floating point
- Character
- Boolean

For better readability of large sized values, Java 7 allows to include '_' in integer literals.

Literal Type	Example
Integer	int x = 10
Octal	int x = 0567
Hexadecimal	int x = 0x9E (to represent number 9E)
Long	long x = 9978547210L;
Binary	byte twelve = 0B1100; (to represent decimal 12)

Using Underscores	int million = 1_000_000;
	int twelve = 0B_1100;
	long multiplier = 12_34_56_78_90_00L;

Float	float x = 0.4f; float y = 1.23F, float z = 0.5e10;
Double	double x = 0.0D; double pi=3.14; double z=9e-9d;
Boolean	boolean member=true; boolean applied=false;
Character	char gender = 'm';
String	String str = "Hello World";
Null	Employee emp = null;

Literals, Assignments, and Variables

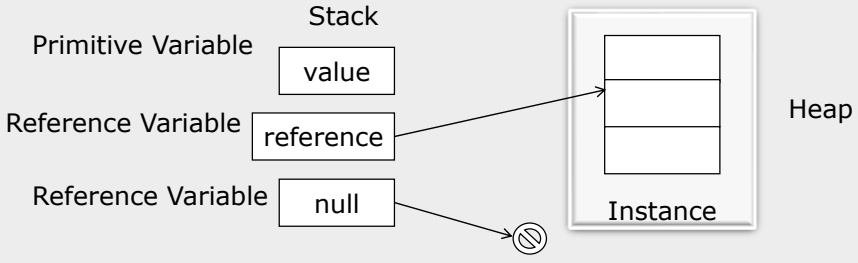
Variables are data placeholders.

Java is a strongly typed language, therefore every variable must have a declared type.

The variables can be of two types:

- reference types: A variable of reference type provides a reference to an object.
- primitive types: A variable of primitive type holds a primitive.

In addition to the data type, a Java variable also has a name or an identifier.



Literal Values for All Primitive Types

Data Type	Default Value (for fields)
byte	0
short	0
int	0
long	0L
float	0.0f
double	0.0d
char	'\u0000'
String (or any object)	null
boolean	false

Arrays can be created for either primitive or reference type elements. Array in java is created as Object using new operator. Once array is created, individual elements can be accessed using index number enclosed in square brackets.

Arrays indexing is zero based, it means the first element of array start at index 0, second element is at 1, and so on. The last element of array is indexed as one minus size of an array.

Array size can be captured by using public final instance variable called length. This feature will avoid any runtime exceptions resulted due to out of bounds access.

Casting Primitive Types

When one type of data is assigned to another type of variable, *automatic type conversion* takes place if:

- Both types are compatible
- Destination type is larger than the source type
- No explicit casting is needed (widening conversion)

```
int a=9; float b; b=a;
```

If there is a possibility of data loss, explicit cast is needed:

```
int i = (int) (5.6/2/7);
```

When one type of data is assigned to another type of variable, an automatic type conversion will take place if the following two conditions are met:
The two types are compatible.
The destination type is larger than the source type.
In this case, a widening conversion takes place. For example, the int type is always large enough to hold all valid byte values, so no explicit cast statement is required.
To widen conversions, numeric types, including integer and floating-point types, are compatible with each other. However, numeric types are not compatible with char or boolean. Also, char and boolean are not compatible with each other.
As mentioned earlier, Java also performs an automatic type conversion when storing a literal integer constant into variables of type byte, short, or long.
Casting Incompatible Types
Automatic type conversions may not fulfill all needs though. For example, assigning an int value to a byte variable. This conversion will not be performed automatically, because a byte is smaller than an int. This is called a narrowing conversion, since you are explicitly making the value narrower so that it will fit into the target type. This is done using a cast - an explicit type conversion. It has this general form: (target-type) value
Here, target-type specifies the desired type to convert the specified value to. For example, the following fragment casts an int to a byte:

```
int i = 125;  byte b;
b = (byte) i;
```

Using a Variable or Array Element That Is Uninitialized and Unassigned

Using Variable that is uninitialized :

```
public  class B {

    public static void main(String[] args) {
        int num;
        System.out.println(num);

    }
}
```

The local variable num may not have been initialized
1 quick fix available:
Initialize variable
Press 'F2' for focus

Using Array that is uninitialized :

```
public  class B {

    public static void main(String[] args) {
        int arr[];
        System.out.println(arr);

    }
}
```

The local variable arr may not have been initialized
1 quick fix available:
Initialize variable
Press 'F2' for focus

Passing Object Reference Variables

```
class Apple{
    public String color="green";
}
public class ParamPassingDemo {

    public static void changeColor(Apple apple){
        apple= new Apple();
        apple.color="red";
    }
    public static void main(String[] args) {
        Apple apple= new Apple();
        System.out.println("color : "+ apple.color);
        changeColor(apple);
        System.out.println("Color : "+apple.color);
    }
}
```

Whenever an object is passed as an argument, an exact copy of the reference variable is created which points to the same location of the object in heap memory as the original reference variable.

As a result of this, whenever we make any change in the same object in the method, that change is reflected in the original object.

However, if we allocate a new object to the passed reference variable, then it won't be reflected in the original object

Does Java Use Pass-By-Value Semantics?

Java is Strictly Pass by Value.

Java manipulates objects by reference, and all object variables are references. However, Java doesn't pass method arguments by reference, but by value.

```
class Apple{
    public String color="red";
}
public class Demo{
public static void main(String[] args) {
    Apple apple = new Apple();
    System.out.println(apple.color);
    changeColor(apple);
    System.out.println(apple.color);
}
public static void changeColor(Apple apple) {
    apple.color="green";
}
}
```

Passing Primitive Variables

Consider the following Java program that passes a primitive type to function.

```
public class Main
{
   public static void main(String[] args)
   {
      int x = 5;
      change(x);
      System.out.println(x);
   }
   public static void change(int x)
   {
      x = 10;
   }
}
```

Declaring Arrays

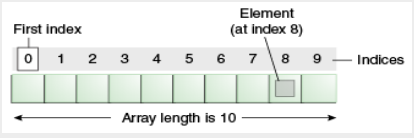
array is a collection of similar type of elements that have contiguous memory location.

Java array is an object the contains elements of similar data type. It is a data structure where we store similar elements. We can store only fixed set of elements in a java array.

Array in java is index based, first element of the array is stored at 0 index.

- int arr [];

  arr = new int[10];



Syntax wherein the array is declared and initialized in the same statement:

```
strWords = { "quiet", "success", "joy", "sorrow", "java" };
```

Constructing Array Objects

Arrays of objects too can be created:

- Example 1:

```
Box barr[] = new Box[3];
barr[0] = new Box();
barr[1] = new Box();
barr[2] = new Box();
```

- Example 2:

```
String[] Words = new String[2];
Words[0]=new  String("Bombay");
Words[1]=new  String("Pune");
```

Use new operator or directly initialize an array. When you create an array object using new, all its slots are initialized for you (0 for numeric arrays, false for boolean, '\0' for character arrays, and null for objects).
Like single dimensional arrays, we can form multidimensional arrays as well. Multidimensional arrays are considered as array of arrays in java and hence can have asymmetrical arrays. See an example next.

Demo

Executing the ArrayDemo.java program

We have seen how to pass parameters to program during compile time using parameter passing. One can pass parameters to a program at runtime too. The args parameter (a String array) in main() receives command line arguments.

```
class ArrayDemo  {
    int intNumbers[];
    ArrayDemo(int i) {
       intNumbers = new int[i];
    }
    void populateArray() {
         for(int i = 0; i < intNumbers.length; ++i) intnumbers[i] = i;
     }
    void displayContents() {
          for(int i = 0; i <intNumbers.length; ++i)
               System.out.println("Number " + i + ": " + intNumbers[i]);
    }
    public static void main(String[] args)               {
        //Accepting array length as command line argument.
         int intArg = Integer.parseInt(args[0]);
         ArrayDemo ad = new ArrayDemo(intArg);
         ad.displayContents();
         ad.populateArray();
         ad.displayContents();
    } }
```

Overview of Wrapper Classes

A Wrapper class is a class whose object wraps or contains a primitive data types. When we create an object to a wrapper class, it contains a field and in this field, we can store a primitive data types. In other words, we can wrap a primitive value into a wrapper class object.

Need of Wrapper Classes

They convert primitive data types into objects. Objects are needed if we wish to modify the arguments passed into a method (because primitive types are passed by value).

The classes in java.util package handles only objects and hence wrapper classes help in this case also.

Data structures in the Collection framework, such as ArrayList and Vector store only objects (reference types) and not primitive types.

Wrapper classes correspond to the primitive data types in the Java language. These classes represent the primitive values as objects. Wrapper objects are immutable. This means that once a wrapper object has a value assigned to it, that value cannot be changed.
Java uses simple or primitive data types, such as int, char and Boolean etc. These data types are not part of the object hierarchy. They are passed by value to methods and cannot be directly passed by reference. However, at times there is a need to create an object representation of these simple data types. Java provides classes that correspond to each of these simple types. These classes encapsulate, or wrap, the simple data type within a class. Thus, they are commonly referred to as wrapper classes. The abstract class Number defines a superclass that is implemented by all numeric wrapper classes.

Wrapper Classes

Simple Data Type	Wrapper Class
byte	Byte
short	Short
int	Integer
long	Long
char	Character
float	Float
double	Double
boolean	Boolean
void	Void

Creating Wrapper Objects

```
Integer a= 10;
Integer b = new Integer(10);
Integer c= Integer.valueOf("123");

Float d= 12.45f;
Float e= new Float(12.12f);
Float f=Float.valueOf(23.32f);
```

Autoboxing

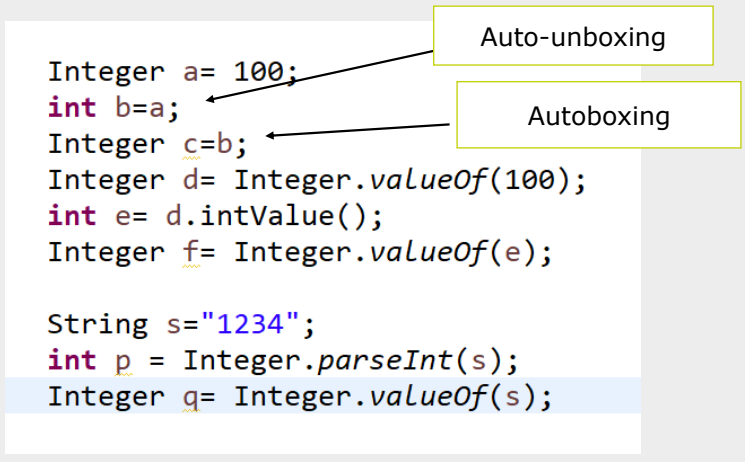
Converting a primitive value into an object of the corresponding wrapper class is called autoboxing.

For example, converting int to Integer class. The Java compiler applies autoboxing when a primitive value is passed as a parameter to a method that expects an object of the corresponding wrapper class.

e.g :

```
Double d = 13.44;
Double e= 13.32;
Double f= e;
```

Using Wrapper Conversion Utilities




```
Integer a= 100;
int b=a;
Integer c=b;
Integer d= Integer.valueOf(100);
int e= d.intValue();
Integer f= Integer.valueOf(e);

String s="1234";
int p = Integer.parseInt(s);
Integer q= Integer.valueOf(s);
```

Garbage Collection

In Java, the programmers don’t need to take care of destroying the objects that are out of use. The Garbage Collector takes care of it.

Garbage Collector is a Daemon thread that keeps running in the background. Basically, it frees up the heap memory by destroying the unreachable objects.

Unreachable objects are the ones that are no longer referenced by any part of the program.

We can choose the garbage collector for our java program through JVM options.

Overview of Memory management and Garbage Collection

Garbage Collection

Garbage collection (GC), also known as *automatic memory management*, is the automatic recycling of dynamically allocated memory (2) Garbage collection is performed by a garbage collector which recycles memory that it can prove will never be used again. Systems and languages which use garbage collection can be described as *garbage-collected.*

All objects are allocated on the heap area managed by the JVM. Every item that the developer uses is treated this way, including class objects, static variables, and even the code itself. As long as an object is being referenced, the JVM considers it alive. Once an object is no longer referenced and therefore is not reachable by the application code, the garbage collector removes it and reclaims the unused memory

Overview of Java's Garbage Collector

Garbage Collector:

- Lowest Priority Daemon Thread
- Runs in the background when JVM starts
- Collects all the unreferenced objects
- Frees the space occupied by these objects
- Call *System.gc()* method to "hint" the JVM to invoke the garbage collector
  - There is no guarantee that it would be invoked. It is implementation dependent

Writing Code That Explicitly Makes Objects Eligible for Garbage Collection

```
class GC{
    protected void finalize() throws Throwable {
        System.out.println("in finalize ");
    }
}
public class GarbageCollectionDemo {
    public static void createObjects(){
        GC obj1= new GC();
        GC obj2= new GC();
        obj1=obj2;
    }
    public static void main(String[] args) {
        createObjects();
        System.gc();
    }
}
```

Summary

In this lesson, you have learnt about:

- Arrays
- Garbage Collection in Java
- Wrapper Classes

Review Question

Question 1: If a display method accepts an integer array and returns nothing , is following call to display method is correct? State true or false.

- display({10,20,30,40,50})

Question 2: All methods in java.util.Arrays class are static (excluding Object class methods).

- True/False