Java

Programming Languages

Lopez, John Ace

Manuben, Geronimo

Montessa, Judy

Quiambao, Lymer

Santiago, Mark Kelvin

BSIT 4-2

HISTORY

Java programming language was originally developed by Sun Microsystems which was initiated by James Gosling and released in 1995 as core component of Sun Microsystems' Java platform (Java 1.0 [J2SE]).

The latest release of the Java Standard Edition is Java SE 8. With the advancement of Java and its widespread popularity, multiple configurations were built to suit various types of platforms. For example: J2EE for Enterprise Applications, J2ME for Mobile Applications.

The new J2 versions were renamed as Java SE, Java EE, and Java ME respectively. Java is guaranteed to be **Write Once, Run Anywhere.**

James Gosling initiated Java language project in June 1991 for use in one of his many set-top box projects. The language, initially called ‘Oak’ after an oak tree that stood outside Gosling's office, also went by the name ‘Green’ and ended up later being renamed as Java, from a list of random words.

Sun released the first public implementation as Java 1.0 in 1995. It promised **Write Once, Run Anywhere** (WORA), providing no-cost run-times on popular platforms.

On 13 November, 2006, Sun released much of Java as free and open source software under the terms of the GNU General Public License (GPL).

On 8 May, 2007, Sun finished the process, making all of Java's core code free and open-source, aside from a small portion of code to which Sun did not hold the copyright.

Features of Java:

- **Object Oriented:** In Java, everything is an Object. Java can be easily extended since it is based on the Object model.
- **Platform Independent:** Unlike many other programming languages including C and C++, when Java is compiled, it is not compiled into platform specific machine, rather into platform

independent byte code. This byte code is distributed over the web and interpreted by the Virtual Machine (JVM) on whichever platform it is being run on.

- **Simple:** Java is designed to be easy to learn. If you understand the basic concept of OOP Java, it would be easy to master.

- **Secure:** With Java's secure feature it enables to develop virus-free, tamper-free systems. Authentication techniques are based on public-key encryption.

- **Architecture-neutral:** Java compiler generates an architecture-neutral object file format, which makes the compiled code executable on many processors, with the presence of Java runtime system.

- **Portable:** Being architecture-neutral and having no implementation dependent aspects of the specification makes Java portable. Compiler in Java is written in ANSI C with a clean portability boundary, which is a POSIX subset.

- **Robust:** Java makes an effort to eliminate error prone situations by emphasizing mainly on compile time error checking and runtime checking.

- **Multithreaded:** With Java's multithreaded feature it is possible to write programs that can perform many tasks simultaneously. This design feature allows the developers to construct interactive applications that can run smoothly.

- **Interpreted:** Java byte code is translated on the fly to native machine instructions and is not stored anywhere. The development process is more rapid and analytical since the linking is an incremental and light-weight process.

- **High Performance:** With the use of Just-In-Time compilers, Java enables high performance.

- **Distributed:** Java is designed for the distributed environment of the internet.

- **Dynamic:** Java is considered to be more dynamic than C or C++ since it is designed to adapt to an evolving environment. Java programs can carry extensive amount of run-time information that can be used to verify and resolve accesses to objects on run-time.

VARIABLES

Local Variables

- Local variables are declared in methods, constructors, or blocks.
- Local variables are created when the method, constructor or block is entered and the variable will be destroyed once it exits the method, constructor, or block.
- Access modifiers cannot be used for local variables.
- Local variables are visible only within the declared method, constructor, or block.
- Local variables are implemented at stack level internally.
- There is no default value for local variables, so local variables should be declared and an initial value should be assigned before the first use.

Example:
Here, *age* is a local variable. This is defined inside *pupAge()* method and its scope is limited to only this method.

```
public class Test{
      public void pupAge(){
            int age = 0;
            age = age + 7;
            System.out.println("Puppy age is : " + age);
      }

      public static void main(String args[]){
            Test test = new Test();
            test.pupAge();
      }
}
```

This will produce the following result:

```
Puppy age is: 7
```

Instance Variables

- Instance variables are declared in a class, but outside a method, constructor or any block.
- When a space is allocated for an object in the heap, a slot for each instance variable value is created.
- Instance variables are created when an object is created with the use of the keyword 'new' and destroyed when the object is destroyed.
- Instance variables hold values that must be referenced by more than one method, constructor or block, or essential parts of an object's state that must be present throughout the class.
- Instance variables can be declared in class level before or after use.
- Access modifiers can be given for instance variables.
- The instance variables are visible for all methods, constructors and block in the class. Normally, it is recommended to make these variables private (access level). However, visibility for subclasses can be given for these variables with the use of access modifiers.
- Instance variables have default values. For numbers, the default value is 0, for Booleans it is false, and for object references it is null. Values can be assigned during the declaration or within the constructor.
- Instance variables can be accessed directly by calling the variable name inside the class. However, within static methods (when instance variables are given accessibility), they should be called using the fully qualified name .*ObjectReference.VariableName*.

Example:

```
import java.io.*;

public class Employee{
      // this instance variable is visible for any child class.
      public String name;

      // salary variable is visible in Employee class only.
      private double salary;

      // The name variable is assigned in the constructor.
      public Employee (String empName){
            name = empName;
      }

      // The salary variable is assigned a value.
      public void setSalary(double empSal){
            salary = empSal;
      }

      // This method prints the employee details.
      public void printEmp(){
            System.out.println("name : " + name );
            System.out.println("salary :" + salary);
      }

      public static void main(String args[]){
            Employee empOne = new Employee("Ransika");
            empOne.setSalary(1000);
            empOne.printEmp();
      }
}
```

This will produce the following result:

```
name : Ransika
salary :1000.0
```

Class/Static Variables

- Class variables also known as static variables are declared with the *static* keyword in a class, but outside a method, constructor or a block.
- There would only be one copy of each class variable per class, regardless of how many objects are created from it.
- Static variables are rarely used other than being declared as constants. Constants are variables that are declared as public/private, final, and static. Constant variables never change from their initial value.
- Static variables are stored in the static memory. It is rare to use static variables other than declared final and used as either public or private constants.
- Static variables are created when the program starts and destroyed when the program stops.
- Visibility is similar to instance variables. However, most static variables are declared public since they must be available for users of the class.
- Default values are same as instance variables. For numbers, the default value is 0; for Booleans, it is false; and for object references, it is null. Values can be assigned during the declaration or within the constructor. Additionally, values can be assigned in special static initializer blocks.
- Static variables can be accessed by calling with the class name *ClassName.VariableName*.
- When declaring class variables as public static final, then variable names (constants) are all in upper case. If the static variables are not public and final, the naming syntax is the same as instance and local variables.

Example:

```
import java.io.*;
public class Employee{
      // salary variable is a private static variable
      private static double salary;

      // DEPARTMENT is a constant
      public static final String DEPARTMENT = "Development ";
      public static void main(String args[]){
            salary = 1000;
            System.out.println(DEPARTMENT + "average salary:" +
salary);
      }
}
```

This will produce the following result:

```
Development average salary:1000
```

DATA TYPES

Variables are nothing but reserved memory locations to store values. This means that when you create a variable you reserve some space in the memory.

Based on the data type of a variable, the operating system allocates memory and decides what can be stored in the reserved memory. Therefore, by assigning different datatypes to variables, you can store integers, decimals, or characters in these variables.

There are two data types available in Java:

- Primitive Datatypes
- Reference/Object Datatypes

Primitive Data Types

There are eight primitive datatypes supported by Java. Primitive datatypes are predefined by the language and named by a keyword. Let us now look into the eight primitive data types in detail.

byte:

- Byte data type is an 8-bit signed two's complement integer
- Minimum value is -128 (-2^7)
- Maximum value is 127 (inclusive)(2^7 -1)
- Default value is 0
- Byte datatype is used to save space in large arrays, mainly in place of integers, since a byte is four times smaller than an integer
- Example: byte a = 100 , byte b = -50

short:

- Short datatype is a 16-bit signed two's complement integer
- Minimum value is -32,768 (-2^15)
- Maximum value is 32,767 (inclusive) (2^15 -1)
- Short datatype can also be used to save memory as byte data type. A short is 2 times smaller than an integer
- Default value is 0
- Example: short s = 10000, short r = -20000

int:

- Int datatype is a 32-bit signed two's complement integer
- Minimum value is - 2,147,483,648 (-2^31)
- Maximum value is 2,147,483,647(inclusive) (2^31 -1)
- Integer is generally used as the default data type for integral values unless there is a concern about memory.
- The default value is 0
- Example: int a = 100000, int b = -200000

long:

- Long datatype is a 64-bit signed two's complement integer
- Minimum value is -9,223,372,036,854,775,808 (-2^63)
- Maximum value is 9,223,372,036,854,775,807 (inclusive) (2^63 -1)
- This type is used when a wider range than int is needed

- Default value is 0L
- Example: long a = 100000L, long b = -200000L

float:
- Float datatype is a single-precision 32-bit IEEE 754 floating point
- Float is mainly used to save memory in large arrays of floating point numbers
- Default value is 0.0f
- Float datatype is never used for precise values such as currency
- Example: float f1 = 234.5f

double:
- Double datatype is a double-precision 64-bit IEEE 754 floating point
- This datatype is generally used as the default data type for decimal values, generally the default choice
- Double datatype should never be used for precise values such as currency
- Default value is 0.0d
- Example: double d1 = 123.4

boolean:
- Boolean datatype represents one bit of information
- There are only two possible values: true and false
- This datatype is used for simple flags that track true/false conditions
- Default value is false
- Example: boolean one = true

char:
- Char datatype is a single 16-bit Unicode character
- Minimum value is '\u0000' (or 0)
- Maximum value is '\uffff' (or 65,535 inclusive)

- Char datatype is used to store any character
- Example: char letterA ='A'

Reference Data Type

- Reference variables are created using defined constructors of the classes. They are used to access objects. These variables are declared to be of a specific type that cannot be changed. For example, Employee, Puppy, etc.
- Class objects and various type of array variables come under reference datatype.
- Default value of any reference variable is null.
- A reference variable can be used to refer any object of the declared type or any compatible type.
- Example: Animal animal = new Animal("giraffe");

Java Literals

A literal is a source code representation of a fixed value. They are represented directly in the code without any computation.

Literals can be assigned to any primitive type variable. For example:

```
byte a = 68;
char a = 'A'
```

byte, int, long, and short can be expressed in decimal(base 10), hexadecimal(base 16) or octal(base 8) number systems as well.

Prefix 0 is used to indicate octal, and prefix 0x indicates hexadecimal when using these number systems for literals. For example:

```
int decimal = 100;
int octal = 0144;
int hexa = 0x64;
```

String literals in Java are specified like they are in most other languages by enclosing a sequence of characters between a pair of double quotes. Examples of string literals are:

```
"Hello World"

"two\nlines"

"\"This is in quotes\""
```

String and char types of literals can contain any Unicode characters. For example:

```
char a = '\u0001';

String a = "\u0001";
```

Java language supports few special escape sequences for String and char literals as well. They are:

Notation	Character Represented
\n	Newline (0x0a)
\r	Carriage Return (0x0d)
\f	Formfeed (0x0c)
\b	Backspace (0x08)
\s	Space (0x20)
\t	Tab
\”	Double Quote
\’	Single Quote
\ddd	Octal Character (ddd)
\uxxxx	Hexadecimal UNICODE Character (xxxx)
\\	Backslash

JAVA OPERATORS

Arithmetic Operators

Arithmetic operators are used in mathematical expressions in the same way that they are used in algebra. The following table lists the arithmetic operators:

Sr. No.	Operator and Example
1	**+ (Addition)** Adds values on either side of the operator. **Example:** A + B will give 30.
2	**- (Subtraction)** Subtracts right-hand operand from left-hand operand **Example:** A - B will give -10
3	*** (Multiplication)** Multiplies values on either side of the operator **Example:** A * B will give 200
4	**/ (Division)** Divides left-hand operand by right-hand operand **Example:** B / A will give 2
5	**% (Modulus)** Divides left-hand operand by right-hand operand and returns remainder **Example:** B % A will give 0
6	**++ (Increment)** Increases the value of operand by 1 **Example:** B++ gives 21
7	**-- (Decrement)** Decreases the value of operand by 1 **Example:** B-- gives 19

Relational Operators

Sr. No.	Operator and Description
1	**== (equal to)** Checks if the values of two operands are equal or not, if yes then condition becomes true. **Example:** (A == B) is true.
2	**!= (not equal to)** Checks if the values of two operands are equal or not if values are not equal then condition becomes true. **Example:** (A !=B) is true.
3	**> (greater than)** Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true. **Example:** (A > B) is not true.
4	**< (less than)** Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true. **Example:** (A < B) is true.
5	**>= (greater than or equal to)** Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true. **Example:** (A >= B) is not true.
6	**<= (less than or equal to)** Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true. **Example:** (A <= B) is true.