1. **What is Java?**

* **History**
* **Features**

1. **JDK, JRE and JVM**
2. **How to set path?**
3. **Hello Java Program**
4. **Compile & Runtime errors**
5. **Command line arguments**
6. **Naming Convention**
7. **Data types**

* **Primitive**
* **Non-Primitive (Wrapper classes)**

1. **Variables & its types**

* **Local**
* **Instance**
* **Static**

1. **Decision Making in Java**

* **Simple if Statement**
* **if…else Statement**
* **if...else if…else statement**
* **Switch statement**

**WHAT IS JAVA?**

* Java is a **programming language.**
* Object oriented programming language.
* Everything is considered as Objects.
* Everything in Java is associated with classes and objects, along with its attributes and methods.
* A class is a group of objects which have common properties.
* An object is an instance of a class. A class is a template or blueprint from which objects are created. So, an object is the instance(result) of a class.
* Java is a high-level language.
* **History**
* Java was developed by Sun Microsystems (which is now the subsidiary of Oracle) in the year 1995.
* James Gosling is known as the father of Java.
* Before Java, its name was Oak. Since Oak was already a registered company, so James Gosling and his team changed the name from Oak to Java.
* **Features**

1. **Simple**

* Java is very easy to learn, and its syntax is simple, clean and easy to understand.
* According to Sun Microsystem, Java language is a simple programming language because:
* Java syntax is based on C++ (so easier for programmers to learn it after C++).
* Java has removed many complicated and rarely-used features, for example, explicit pointers, operator overloading, etc.
* There is no need to remove unreferenced objects because there is an Automatic Garbage Collection in Java.

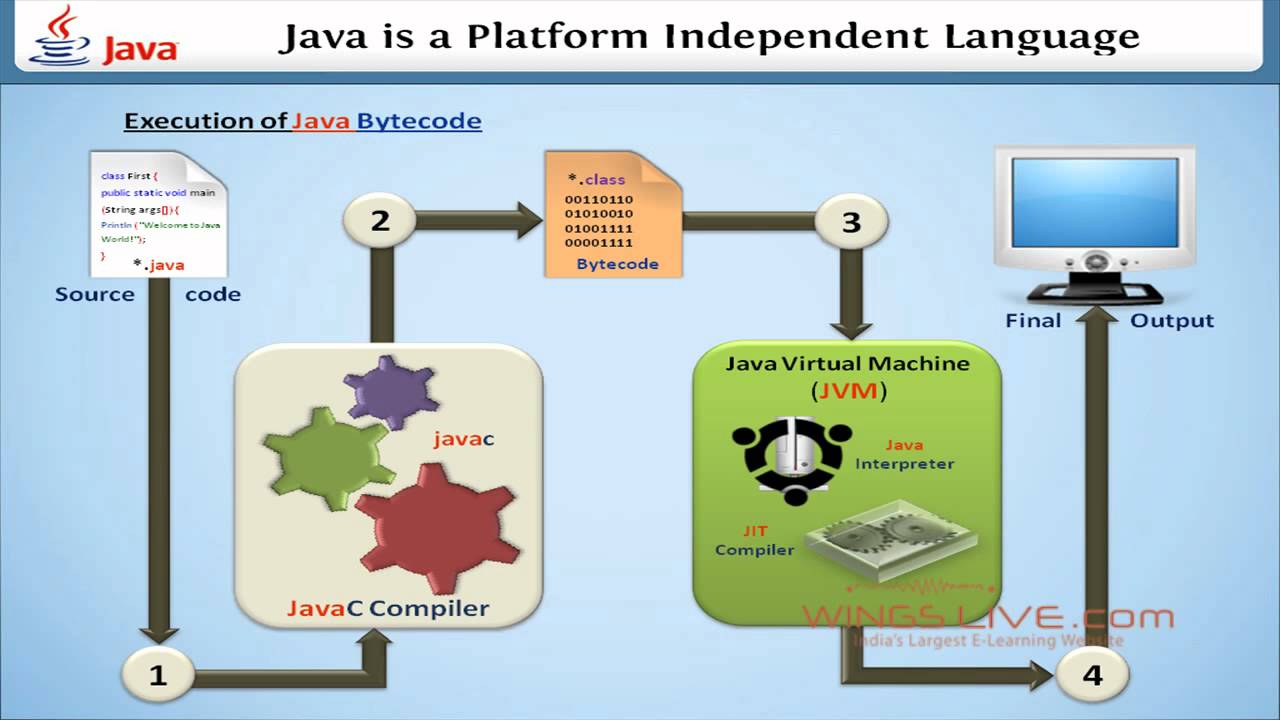
1. **Object Oriented**

* Java is an object-oriented programming language.
* Everything in Java is an object.
* Object-oriented means we organize our software as a combination of different types of objects that incorporate both data and behaviour.
* OOPs is a methodology that simplifies software development and maintenance by providing some rules.
* Basic concepts of OOPs are:

Object, Class, Inheritance, Polymorphism, Abstraction & Encapsulation

1. **Platform Independent**

* Languages like C, C++ compiled into platform specific machines.
* While Java is a write once, run anywhere language.
* A platform is the hardware or software environment in which a program runs.

****

1. **Secured**

* Java is best known for its security.
* With Java, we can develop virus free systems.
* Java is secured because:
* No explicit pointer
* Java Programs run inside a virtual sandbox.

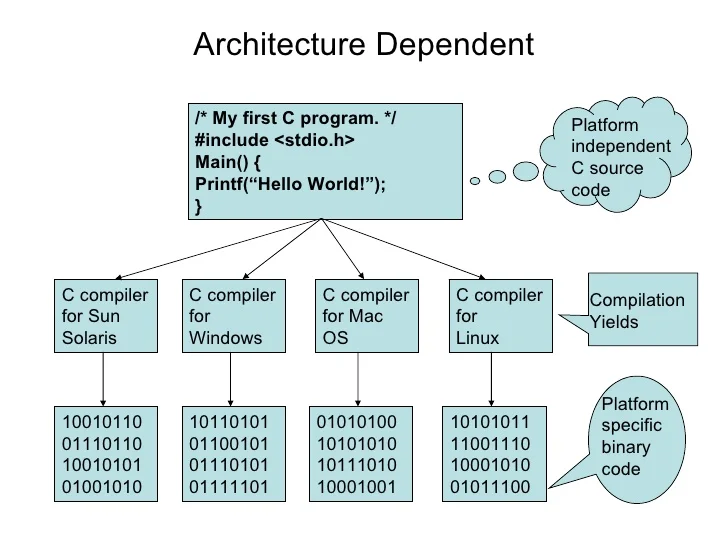
****

1. **Robust**

* The English mining of Robust is strong. Java is robust because:
* It uses strong memory management.
* There is a lack of pointers that avoids security problems.
* Java provides automatic garbage collection which runs on the Java Virtual Machine to get rid of objects which are not being used by a Java application anymore.
* There are exception handling and the type checking mechanism in Java. All these points make Java robust.

1. **Architecture-neutral**

* No implementation dependent features, for example, the size of primitive types is fixed.
* In C programming, int data types occupies
* 2 bytes of memory for 32-bit architecture
* 4 bytes of memory for 64-bit architecture
* 4 bytes of memory for
* Both 32 and 64-bit architecture in Java.

****

1. **Portable**

* Java is portable because it facilitates you to carry the Java bytecode to any platform.
* It doesn't require any implementation.

1. **High-performance**

* Java is faster than other traditional interpreted programming languages.
* Because Java bytecode is "close" to native code.
* It is still a little bit slower than a compiled language (e.g., C++).
* Java is an interpreted language that is why it is slower than compiled languages, e.g., C, C++, etc.

1. **Distributed**

* Java is distributed because it facilitates users to create distributed applications in Java
* RMI and EJB are used for creating distributed applications.
* This feature of Java makes us able to access files by calling the methods from any machine on the internet.

1. **Multi-threaded**

* A thread is like a separate program, executing concurrently.
* We can write Java programs that deal with many tasks at once by defining multiple threads.
* The main advantage of multi-threading is that it doesn’t occupy memory for each thread.
* It shares a common memory area.
* Threads are important for multi-media, Web applications, etc.

1. **Dynamic**

* Java is a dynamic language.
* It supports the dynamic loading of classes.
* It means classes are loaded on demand.
* It also supports functions from its native languages, i.e., C and C++.

**JVM, JRE and JDK**

* **JVM**
* JVM (Java Virtual Machine) is an abstract machine.
* It is called a virtual machine because it doesn't physically exist.
* It is a specification that provides a runtime environment in which Java bytecode can be executed.
* It can also run those programs which are written in other languages and compiled to Java bytecode.
* The JVM performs the following main tasks:
* Loads code
* Verifies code
* Executes code
* Provides runtime environment
* **JRE**
* JRE is an acronym for Java Runtime Environment.
* It is also written as Java RTE.
* The Java Runtime Environment is a set of software tools which are used for developing Java applications.
* It is used to provide the runtime environment.
* It is the implementation of JVM. It physically exists. It contains a set of libraries + other files that JVM uses at runtime.

****

* **JDK**
* JDK is an acronym for Java Development Kit.
* The Java Development Kit (JDK) is a software development environment which is used to develop Java applications and applets.
* It physically exists. It contains JRE + development tools.
* The JDK contains a private Java Virtual Machine (JVM) and a few other resources such as an interpreter/loader (java), a compiler (javac), an archiver (jar), a documentation generator (Javadoc), etc. to complete the development of a Java Application.

****

**HOW TO SET PATH?**

* To set the path of JDK, following steps:

1. Open the Control Panel by clicking the Start button and searching for “Control Panel”.
2. In the Control Panel, click on “System and Security” and then click on “System”.
3. Click on “Advanced system settings” in the left sidebar.
4. In the System Properties dialog box, click on the “Environment Variables” button.
5. In the Environment Variables dialog box, under “System variables”, locate the “Path” variable and click on “Edit”.
6. In the “Edit Environment Variable” dialog box, click on “New” and enter the path of your JDK installation folder.

For example, “C:\Program Files\Java\jdk-16.0.1\bin”.

1. Click “OK” to save the changes.
2. Close all the dialog boxes and open a new command prompt or terminal window.
3. Type the following command to verify that Java is working:

java -version

**HELLO JAVA PROGRAM**

**public class HelloWorld**

**{**

**public static void main(String[] args)**

**{**

**System.out.println("Hello, World!");**

**}**

**}**

**COMPILE & RUNTIME ERRORS**

**Compile errors**

* Compile-time errors occur when there are syntactical issues present in application code.

For example, missing semicolons or parentheses, misspelled keywords or usage of undeclared variables.

* These syntax errors are detected by the Java compiler at compile-time and an error message is displayed on the screen.
* The compiler prevents the code from being executed until the error is fixed. Therefore, these errors must be addressed by debugging before the program can be successfully run.

**Runtime errors**

* A runtime error in Java is an application error that occurs during the execution of a program.
* A runtime error occurs when a program is syntactically correct but contains an issue that is only detected during program execution.
* These issues cannot be caught at compile-time by the Java compiler and are only detected by the Java Virtual Machine (JVM) when the application is running.

The most common causes of runtime errors in Java are:

* Dividing a number by zero.
* Accessing an element in an array that is out of range.
* Attempting to store an incompatible type value to a collection.
* Passing an invalid argument to a method.
* Attempting to convert an invalid string to a number.
* Insufficient space in memory for thread data.

**COMMAND LINE ARGUMENTS**

* The java command-line argument is an argument i.e. passed at the time of running the java program.
* The arguments passed from the console can be received in the java program and it can be used as an input.
* So, it provides a convenient way to check the behavior of the program for the different values. You can pass **N** (1,2,3 and so on) numbers of arguments from the command prompt.

Simple example of command-line argument in java:

**class** CommandLineExample

{

**public** **static** **void** main(String args[])

{

System.out.println("Your first argument is: "+args[0]);

}

}

**NAMING CONVENTION**

* Java naming convention is a rule to follow as you decide what to name your identifiers such as class, package, variable, constant, method, etc.
* Naming conventions are guidelines for naming various elements in your code, such as classes, variables, methods, packages, and more.
* Following these conventions makes your code more readable and maintainable and helps improve code consistency across projects.

Here are some common Java naming conventions with examples:

* Class Names (CamelCase):

Class names should start with an uppercase letter, and each subsequent word should also start with an uppercase letter (CamelCase).

Example: MyClass, StudentRecord, Calculator

* Method Names (camelCase):

Method names should start with a lowercase letter, and each subsequent word should start with an uppercase letter (camelCase).

Example: calculateTotal, getStudentName, printReport

* Variable Names (camelCase):

Variable names should start with a lowercase letter, and each subsequent word should start with an uppercase letter (camelCase).

Example: firstName, studentAge, totalAmount

* Constants (UPPERCASE):

Constants should be in all uppercase letters with words separated by underscores.

Example: MAX\_VALUE, PI, DEFAULT\_COLOR

* Packages (lowercase):

Package names should be in all lowercase letters.

Example: com.example.myapp, org.openai.nlp

* Interfaces (CamelCase):

Interface names should follow the same convention as class names (CamelCase).

Example: Shape, DatabaseConnection

* Method Parameters (camelCase):

Method parameters should follow the same convention as variable names (camelCase).

Example: calculateArea(int length, int width)

**DATA TYPES**

Data types specify the different sizes and values that can be stored in the variable. There are two types of data types in Java:

* **Primitive** 
  + A primitive data type specifies the size and type of variable values, and it has no additional methods.

There are 8 types of primitive data types:

* boolean data type
* byte data type
* char data type
* short data type
* int data type
* long data type
* float data type
* double data type

|  |  |  |
| --- | --- | --- |
| **Data Type** | **Size** | **Description** |
| byte | 1 byte | Stores whole numbers from -128 to 127 |
| short | 2 bytes | Stores whole numbers from -32,768 to 32,767 |
| int | 4 bytes | Stores whole numbers from -2,147,483,648 to 2,147,483,647 |
| long | 8 bytes | Stores whole numbers from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807 |
| float | 4 bytes | Stores fractional numbers. Sufficient for storing 6 to 7 decimal digits |
| double | 8 bytes | Stores fractional numbers. Sufficient for storing 15 decimal digits |
| boolean | 1 bit | Stores true or false values |
| char | 2 bytes | Stores a single character/letter or ASCII values |

* **Non-Primitive (Wrapper classes)** 
  + The non-primitive data types include [Classes](https://www.javatpoint.com/object-and-class-in-java), [Interfaces](https://www.javatpoint.com/interface-in-java), and [Arrays](https://www.javatpoint.com/array-in-java).

Non-primitive data types are called reference types because they refer to objects.

* The main difference between primitive and non-primitive data types are:
* Primitive types are predefined (already defined) in Java. Non-primitive types are created by the programmer and is not defined by Java (except for String).
* Non-primitive types can be used to call methods to perform certain operations, while primitive types cannot.
* A primitive type has always a value, while non-primitive types can be null.
* A primitive type starts with a lowercase letter, while non-primitive types starts with an uppercase letter.

**VARIABLES & ITS TYPES**

* A variable is a container which holds the value while the [Java program](https://www.javatpoint.com/simple-program-of-java) is executed. A variable is assigned with a data type.
* Variable is a name of memory location.
* A variable is the name of a reserved area allocated in memory. In other words, it is a name of the memory location. It is a combination of "vary + able" which means its value can be changed.

There are three types of variables in java:

* **Local**
* A variable declared inside the body of the method is called local variable. You can use this variable only within that method and the other methods in the class aren't even aware that the variable exists.
* A local variable cannot be defined with "static" keyword.
* **Instance**
* A variable declared inside the class but outside the body of the method, is called an instance variable. It is not declared as [static](https://www.javatpoint.com/static-keyword-in-java).
* It is called an instance variable because its value is instance-specific and is not shared among instances.
* **Static**
* A variable that is declared as static is called a static variable.
* It cannot be local.
* We can create a single copy of the static variable and share it among all the instances of the class.
* Memory allocation for static variables happens only once when the class is loaded in the memory.

**DECISION MAKING IN JAVA**

Decision-making statements decide which statement to execute and when.

Decision-making statements evaluate the Boolean expression and control the program flow depending upon the result of the condition provided.

* **Simple if Statement**

It is the most basic statement among all control flow statements in Java. It evaluates a Boolean expression and enables the program to enter a block of code if the expression evaluates to true.

**Syntax:**

1. **if**(condition) {
2. statement 1; //executes when condition is true
3. }

* **if…else Statement**

The [if-else statement](https://www.javatpoint.com/java-if-else) is an extension to the if-statement, which uses another block of code, i.e., else block. The else block is executed if the condition of the if-block is evaluated as false.

**Syntax:**

1. **if**(condition) {
2. statement 1; //executes when condition is true
3. }
4. **else**{
5. statement 2; //executes when condition is false
6. }

* **if...else if…else statement**

The if-else-if statement contains the if-statement followed by multiple else-if statements.

In other words, we can say that it is the chain of if-else statements that create a decision tree where the program may enter in the block of code where the condition is true. We can also define an else statement at the end of the chain.

**Syntax:**

1. **if**(condition 1) {
2. statement 1; //executes when condition 1 is true
3. }
4. **else** **if**(condition 2) {
5. statement 2; //executes when condition 2 is true
6. }
7. **else** {
8. statement 2; //executes when all the conditions are false
9. }

* **Switch statement**

In Java, [Switch statements](https://www.javatpoint.com/java-switch) are similar to if-else-if statements.

The switch statement contains multiple blocks of code called cases and a single case is executed based on the variable which is being switched.

The switch statement is easier to use instead of if-else-if statements. It also enhances the readability of the program.

**Syntax:**

1. **switch** (expression){
2. **case** value1:
3. statement1;
4. **break**;
5. .
6. .
7. .
8. **case** valueN:
9. statementN;
10. **break**;
11. **default**:
12. **default** statement;
13. }