History of Java

The history of Java is very interesting. Java was originally designed for interactive television, but it was too advanced technology for the digital cable television industry at the time. The history of java starts from Green Team. Java team members (also known as Green Team), initiated this project to develop a language for digital devices such as set-top boxes, televisions etc. But, it was suited for internet programming. Later, Java technology was incorporated by Netscape. The principles for creating Java programming were "Simple, Robust, Portable, Platform-independent, Secured, High Performance, Multithreaded, Architecture Neutral, Object-Oriented, Interpreted and Dynamic". Currently, Java is used in internet programming, mobile devices, games, e-business solutions etc. There are given the major points that describes the history of java.

1) James Gosling, Mike Sheridan, and Patrick Naughton initiated the Java language project in June 1991.
2) The small team of sun engineers called Green Team.
3) Originally designed for small, embedded systems in electronic appliances like set-top boxes.
4) Firstly, it was called "Greentalk" by James Gosling and file extension was .gt. After that, it was called Oak and was developed as a part of the Green project.

Why Java named as "Oak"

5) Why Oak? Oak is a symbol of strength and choosen as a national tree of many countries like U.S.A., France, Germany, Romania etc.
6) In 1995, Oak was renamed as "Java" because it was already a trademark by Oak Technologies.

Why Java Programming named as "Java"

7) Why had they choosen java name for java language? The team gathered to choose a new name. The suggested words were "dynamic", "revolutionary", "Silk", "jolt", "DNA" etc. They wanted something that reflected the essence of the technology: revolutionary, dynamic, lively, cool, unique, and easy to spell and fun to say. According to James Gosling "Java was one of the top choices along with Silk". Since java was so unique, most of the team members preferred java.
8) Java is an island of Indonesia where first coffee was produced (called java coffee).
9) Notice that Java is just a name not an acronym.
10) Originally developed by James Gosling at Sun Microsystems (which is now a subsidiary of Oracle Corporation) and released in 1995.
11) In 1995, Time magazine called Java one of the Ten Best Products of 1995.
12) JDK 1.0 released in(January 23, 1996).

Java Architecture

Java is much more than just a popular programming language instead it's a complete architecture. The entire Java archirecture can be broadly categorized into three technologies which work together to make Java a powerful software development architecture. The components of the Java architecture are:-

- **The Java Programming Language** - JLS (Java Langauge Specifications) provide the syntax, constructs, grammar, etc. of the Java language.
- **The Java APIs** - Java Application Programming Interfaces (APIs) make available plethora of functionalities to the application developers so that they can focus on their core business functionality without bothering about the common tasks including those from the areas - I/O, Network, Language Constructs, Multithreading, Serialization, etc. Many vendors have implemented these APIs including the standard implementation given by Sun Microsystems which is normally used as a benchmark. The Java APIs are simply a collection of Java .class files. A .class file is the bytecode representation of a valid Java program. Bytecodes are a collection operator-operand tuples which are converted into the native machine level instructions by the underlying JVM.
- **The Java Virtual Machine** - The JVM is the abstract machine which runs all the bytecodes on a particular machine. JVM converts the bytecodes into native instructions which are executed by the underlying Operating System

Java Class File

A **Java class file** is a file containing Java bytecode and having **.class extension** that can be executed by JVM. A Java class file is created by a Java compiler from *.java* files as a result of successful compilation. As we know that a single Java programming language source file (*or we can say .java file*) may contain one class or more than one class. So if a.*java* file has more than one class then each class will compile into a separate class files.
For Example: Save this below code as **Test.java** on your system.

```
// Compiling this Java program would
// result in multiple class files.

class Sample
{

}

// Class Declaration
class Student
{

}
// Class Declaration
class Test
{
    public static void main(String[] args)
    {
        System.out.println("Class File Structure");
    }
}
```

For Compiling:
javac Test.java

After compilation there will be **3 class** files in corresponding folder named as:

- Sample.class
- Student.class
- Test.class

A **a single class file structure** contains attributes that describe a class file.

Representation of Class File Structure

```
ClassFile
{
    magic_number;
    minor_version;
    major_version;
    constant_pool_count;
    constant_pool[];
    access_flags;
    this_class;
    super_class;
    interfaces_count;
    interfaces[];
    fields_count;
    fields[];
    methods_count;
    methods[];
    attributes_count;
    attributes[];
}
```

Java Runtime Environment (JRE)

What does Java Runtime Environment (JRE)mean?

The Java Runtime Environment (JRE) is a set of software tools for development of Java applications. It combines the Java Virtual Machine (JVM), platform core classes and supporting libraries.

JRE is part of the Java Development Kit (JDK), but can be downloaded separately. JRE was originally developed by Sun Microsystems Inc., a wholly-owned subsidiary of Oracle Corporation.

Also known as Java runtime.

Java Runtime Environment (JRE)

What does JRE consists of ?

JRE consists of the following components:

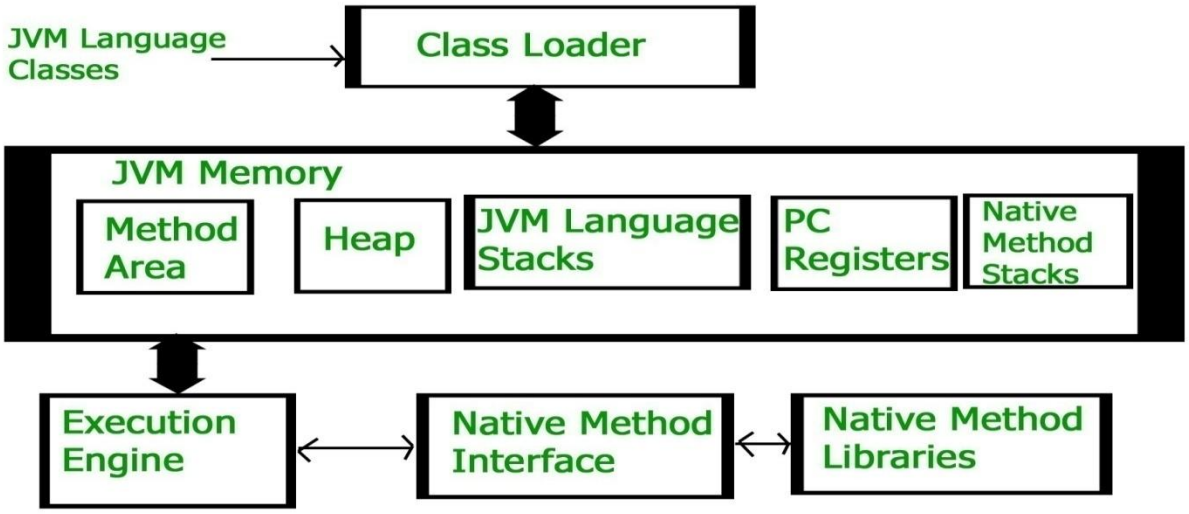
- **Deployment technologies**, including deployment, Java Web Start and Java Plug-in.
- **User interface toolkits**, including *Abstract Window Toolkit (AWT), Swing, Java 2D, Accessibility, Image I/O, Print Service, Sound, drag and drop (DnD)* and *input methods*.
- **Integration libraries**, including *Interface Definition Language (IDL), Java Database Connectivity (JDBC), Java Naming and Directory Interface (JNDI), Remote Method Invocation (RMI), Remote Method Invocation Over Internet Inter-Orb Protocol (RMI-IIOP)* and *scripting*.
- **Other base libraries**, including *international support, input/output (I/O), extension mechanism, Beans, Java Management Extensions (JMX), Java Native Interface (JNI), Math, Networking, Override Mechanism, Security, Serialization* and *Java for XML Processing (XML JAXP)*.
- **Lang and util base libraries**, including *lang and util, management, versioning, zip, instrument, reflection, Collections, Concurrency Utilities, Java Archive (JAR), Logging, Preferences API, Ref Objects* and *Regular Expressions*.
- **Java Virtual Machine (JVM)**, including *Java HotSpot Client* and *Server Virtual Machines*.

How JVM Works – JVM Architecture?

JVM(Java Virtual Machine) acts as a run-time engine to run Java applications. JVM is the one that actually calls the **main** method present in a java code. JVM is a part of JRE(Java Runtime Environment).

Java applications are called WORA (Write Once Run Anywhere). This means a programmer can develop Java code on one system and can expect it to run on any other Java enabled system without any adjustment. This is all possible because of JVM.

When we compile a *.java* file, *.class* files(contains byte-code) with the same class names present in *.java* file are generated by the Java compiler. This *.class* file goes into various steps when we run it. These steps together describe the whole JVM.



Class Loader Subsystem
It is mainly responsible for three activities.

- Loading
- Linking
- Initialization

Loading :
The Class loader reads the *.class* file, generate the corresponding binary data and save it in method area. For each *.class* file, JVM stores following information in method area.

- Fully qualified name of the loaded class and its immediate parent class.
- Whether *.class* file is related to Class or Interface or Enum
- Modifier, Variables and Method information etc.

After loading *.class* file, JVM creates an object of type Class to represent this file in the heap memory. Please note that this object is of type Class predefined in *java.lang* package. This Class object can be used by the programmer for getting class level information like name of class, parent name, methods and variable information etc. To get this object reference we can use *getClass()* method of Object class.

Linking :
Performs verification, preparation, and (optionally) resolution.

- *Verification* : It ensures the correctness of *.class* file i.e. it check whether this file is properly formatted and generated by valid compiler or not. If verification fails, we get run-time exception *java.lang.VerifyError*.
- *Preparation* : JVM allocates memory for class variables and initializing the memory to default values.
- *Resolution* : It is the process of replacing symbolic references from the type with direct references. It is done by searching into method area to locate the referenced entity.

Initialization :
In this phase, all static variables are assigned with their values defined in the code and static block(if any). This is executed from top to bottom in a class and from parent to child in class hierarchy.
In general, there are three class loaders :

- *Bootstrap class loader* : Every JVM implementation must have a bootstrap class loader, capable of loading trusted classes. It loads core java API classes present in*JAVA_HOME/jre/lib* directory. This path is popularly known as bootstrap path. It is implemented in native languages like C, C++.
- *Extension class loader* : It is child of bootstrap class loader. It loads the classes present in the extensions directories *JAVA_HOME/jre/lib/ext*(Extension path) or any other directory specified by the java.ext.dirs system property. It is implemented in java by the *sun.misc.Launcher$ExtClassLoader* class.
- *System/Application class loader* : It is child of extension class loader. It is responsible to load classes from application class path. It internally uses Environment Variable which mapped to java.class.path. It is also implemented in Java by the*sun.misc.Launcher$AppClassLoader* class.

JVM Memory

Method area : In method area, all class level information like class name, immediate parent class name, methods and variables information etc. are stored, including static variables. There is only one method area per JVM, and it is a shared resource.

- **Heap area :**Information of all objects is stored in heap area. There is also one Heap Area per JVM. It is also a shared resource.
- **Stack area :**For every thread, JVM create one run-time stack which is stored here. Every block of this stack is called activation record/stack frame which store methods calls. All local variables of that method are stored in their corresponding frame. After a thread terminate, it's run-time stack will be destroyed by JVM. It is not a shared resource.
- **PC Registers :**Store address of current execution instruction of a thread. Obviously each thread has separate PC Registers.
- **Native method stacks :**For every thread, separate native stack is created. It stores native method information.

Execution Engine

Execution engine execute the *.class* (bytecode). It reads the byte-code line by line, use data and information present in various memory area and execute instructions. It can be classified in three parts :-

- *Interpreter* : It interprets the bytecode line by line and then executes. The disadvantage here is that when one method is called multiple times, every time interpretation is required.
- *Just-In-Time Compiler(JIT)* : It is used to increase efficiency of interpreter.It compiles the entire bytecode and changes it to native code so whenever interpreter see repeated method calls,JIT provide direct native code for that part so re-interpretation is not required,thus efficiency is improved.
- *Garbage Collector* : It destroy un-referenced objects.For more on Garbage Collector,refer Garbage Collector.

Java Native Interface (JNI) :

It is a interface which interacts with the Native Method Libraries and provides the native libraries(C, C++) required for the execution. It enables JVM to call C/C++ libraries and to be called by C/C++ libraries which may be specific to hardware.

Native Method Libraries :

It is a collection of the Native Libraries(C, C++) which are required by the Execution Engine.

JAVA API

java.applet	Provides the classes necessary to create an applet and the classes an applet uses to communicate with its applet context.
java.awt	Contains all of the classes for creating user interfaces and for painting graphics and images.
java.awt.color	Provides classes for color spaces.
java.awt.datatransfer	Provides interfaces and classes for transferring data between and within applications.
java.awt.dnd	Drag and Drop is a direct manipulation gesture found in many Graphical User Interface systems that provides a mechanism to transfer information between two entities logically associated with presentation elements in the GUI.
java.awt.event	Provides interfaces and classes for dealing with different types of events fired by AWT components.
java.awt.font	Provides classes and interface relating to fonts.
java.awt.geom	Provides the Java 2D classes for defining and performing operations on Objects related to two-dimensional geometry.
java.awt.im	Provides classes and interfaces for the input method framework.
java.awt.im.spi	Provides interfaces that enable the development of input methods that can be used with any Java runtime environment.
java.awt.image	Provides classes for creating and modifying images.
java.awt.image.renderable	Provides classes and interfaces for producing rendering-independent images.
java.awt.print	Provides classes and interfaces for a general printing API.
java.beans	Contains classes related to developing *beans* -- components based on the JavaBeans™ architecture.
java.io	Provides for system input and output through data streams, serialization and the file system.
java.lang	Provides classes that are fundamental to the design of the Java programming language.
java.math	Provides classes for performing arbitrary-precision integer arithmetic (BigInteger) and arbitrary-precision decimal arithmetic (BigDecimal).
java.net	Provides the classes for implementing networking applications.
java.nio	Defines buffers, which are containers for data, and provides an overview of the other NIO packages.
java.nio.channels	Defines channels, which represent connections to entities that are capable of performing I/O operations, such as files and sockets; defines selectors, for multiplexed, non-blocking I/O operations.
java.nio.channels.spi	Service-provider classes for the **java.nio.channels** package.
java.nio.charset	Defines charsets, decoders, and encoders, for translating between bytes and

	Unicode characters.
java.nio.charset.spi	Service-provider classes for the **java.nio.charset** package.
java.nio.file	Defines interfaces and classes for the Java virtual machine to access files, file attributes, and file systems.
java.nio.file.attribute	Interfaces and classes providing access to file and file system attributes.
java.nio.file.spi	Service-provider classes for the **java.nio.file** package.
java.rmi	Provides the RMI package.
java.rmi.activation	Provides support for RMI Object Activation.
java.rmi.dgc	Provides classes and interface for RMI distributed garbage-collection (DGC).
java.rmi.registry	Provides a class and two interfaces for the RMI registry.
java.rmi.server	Provides classes and interfaces for supporting the server side of RMI.
java.security	Provides the classes and interfaces for the security framework.
java.security.acl	The classes and interfaces in this package have been superseded by classes in the java.security package.
java.security.cert	Provides classes and interfaces for parsing and managing certificates, certificate revocation lists (CRLs), and certification paths.
java.security.interfaces	Provides interfaces for generating RSA (Rivest, Shamir and Adleman Asymmetric Cipher algorithm) keys as defined in the RSA Laboratory Technical Note PKCS#1, and DSA (Digital Signature Algorithm) keys as defined in NIST's FIPS-186.
java.security.spec	Provides classes and interfaces for key specifications and algorithm parameter specifications.
java.sql	Provides the API for accessing and processing data stored in a data source (usually a relational database) using the Java™programming language.
java.util	Contains the collections framework, legacy collection classes, event model, date and time facilities, internationalization, and miscellaneous utility classes (a string tokenizer, a random-number generator, and a bit array).
java.util.concurrent	Utility classes commonly useful in concurrent programming.
java.util.concurrent.atomic	A small toolkit of classes that support lock-free thread-safe programming on single variables.
java.util.concurrent.locks	Interfaces and classes providing a framework for locking and waiting for conditions that is distinct from built-in synchronization and monitors.
java.util.jar	Provides classes for reading and writing the JAR (Java ARchive) file format, which is based on the standard ZIP file format with an optional manifest file.
java.util.logging	Provides the classes and interfaces of the Java™ 2 platform's core logging facilities.
java.util.prefs	This package allows applications to store and retrieve user and system preference and configuration data.
java.util.regex	Classes for matching character sequences against patterns specified by regular

	expressions.
java.util.spi	Service provider classes for the classes in the java.util package.
java.util.zip	Provides classes for reading and writing the standard ZIP and GZIP file formats.

Java Development Kit (JDK)

Definition - What does Java Development Kit (JDK) mean ?

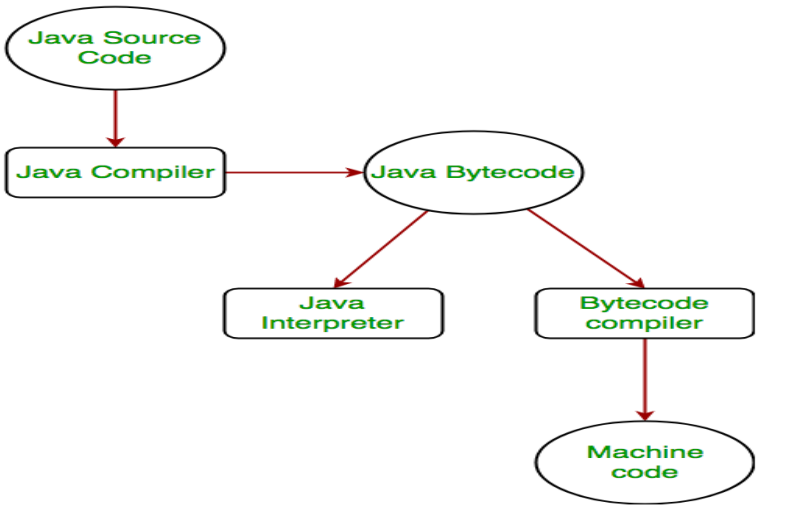
The Java Development Kit (JDK) is a software development environment used for developing Java applications and applets. It includes the Java Runtime Environment (JRE), an interpreter/loader (java), a compiler (javac), an archiver (jar), a documentation generator (javadoc) and other tools needed in Java development.

How is Java platform independent ?

The meaning of platform independent is that, the java source code can run on all operating systems.
A program is written in a language which is a human readable language. It may contain words, phrases etc which the machine does not understand. For the source code to be understood by the machine, it needs to be in a language understood by machines, typically a machine-level language. So, here comes the role of a compiler. The compiler converts the high-level language (human language) into a format understood by the machines. Therefore, a compiler is a program that translates the source code for another program from a programming language into executable code.
This executable code may be a sequence of machine instructions that can be executed by the CPU directly, or it may be an intermediate representation that is interpreted by a virtual machine. This intermediate representation in Java is the **Java Byte Code.**

Step by step Execution of Java Program:

- Whenever, a program is written in JAVA, the javac compiles it.
- The result of the JAVA compiler is the **.class file or the bytecode** and not the machine native code (unlike C compiler).
- The bytecode generated is a non-executable code and needs an interpreter to execute on a machine. This interpreter is the JVM and thus the Bytecode is executed by the JVM.
- And finally program runs to give the desired output.



In case of C or C++ (language that are not platform independent), the compiler generates an .exe file which is OS dependent. When we try to run this .exe file on another OS it does not run, since it is OS dependent and hence is not compatible with the other OS.

Java Lambda Expressions

Lambda expression is a new and important feature of Java which was included in Java SE 8. It provides a clear and concise way to represent one method interface using an expression. It is very useful in collection library. It helps to iterate, filter and extract data from collection.

The Lambda expression is used to provide the implementation of an interface which has functional interface. It saves a lot of code. In case of lambda expression, we don't need to define the method again for providing the implementation. Here, we just write the implementation code.

Java lambda expression is treated as a function, so compiler does not create .class file.

Functional Interface

Lambda expression provides implementation of *functional interface*. An interface which has only one abstract method is called functional interface. Java provides an anotation *@FunctionalInterface*, which is used to declare an interface as functional interface.

Why use Lambda Expression

1. To provide the implementation of Functional interface.
2. Less coding.

Java Lambda Expression Syntax

(argument-list) -> {body}

Java lambda expression is consisted of three components.

1) Argument-list: It can be empty or non-empty as well.

2) Arrow-token: It is used to link arguments-list and body of expression.

3) Body: It contains expressions and statements for lambda expression.

Let's see a scenario. If we don't implement Java lambda expression. Here, we are implementing an interface method without using lambda expression.

Without Lambda Expression

```
1. interface Drawable{
2.     public void draw();
3. }
4. public class LambdaExpressionExample {
5.     public static void main(String[] args) {
6.         int width=10;
7.
8.         //without lambda, Drawable implementation using anonymous class
9.         Drawable d=new Drawable(){
10.            public void draw(){System.out.println("Drawing "+width);}
11.        };
12.        d.draw();
13.    }
14. }
```

Output:

```
Drawing 10
```

Java Lambda Expression Example

Now, we are implementing the above example with the help of lambda expression.

```
1. @FunctionalInterface  //It is optional
2. interface Drawable{
3.     public void draw();
4. }
5.
6. public class LambdaExpressionExample2 {
7.     public static void main(String[] args) {
8.         int width=10;
9.
10.        //with lambda
11.        Drawable d2=()->{
12.            System.out.println("Drawing "+width);
13.        };
14.        d2.draw();
15.    }
16. }
```

Output:

```
Drawing 10
```

Java Method References

Java provides a new feature called method reference in Java 8. Method reference is used to refer method of functional interface. It is compact and easy form of lambda expression. Each time when you are using lambda expression to just referring a method, you can replace your lambda expression with method reference. In this tutorial, we are explaining method reference concept in detail.

Types of Method References

There are following types of method references in java:

1. Reference to a static method.
2. Reference to an instance method.
3. Reference to a constructor.

1) Reference to a Static Method

You can refer to static method defined in the class. Following is the syntax and example which describe the process of referring static method in Java.

Syntax

ContainingClass::staticMethodName

Example 1

In the following example, we have defined a functional interface and referring a static method to it's functional method say().

```
1. interface Sayable{
2.     void say();
3. }
4. public class MethodReference {
5.     public static void saySomething(){
6.         System.out.println("Hello, this is static method.");
7.     }
8.     public static void main(String[] args) {
9.         // Referring static method
10.        Sayable sayable = MethodReference::saySomething;
11.        // Calling interface method
12.        sayable.say();
13.    }
14. }
```

Output:

```
Hello, this is static method.
```

2) Reference to an Instance Method

like static methods, you can refer instance methods also. In the following example, we are describing the process of referring the instance method.

Syntax

containingObject::instanceMethodName

Example 1

In the following example, we are referring non-static methods. You can refer methods by class object and anonymous object.

```
1. interface Sayable{
2.     void say();
3. }
4. public class InstanceMethodReference {
5.     public void saySomething(){
6.         System.out.println("Hello, this is non-static method.");
7.     }
8.     public static void main(String[] args) {
9.         InstanceMethodReference methodReference = new InstanceMethodReference(); // Creating
    object
10.        // Referring non-static method using reference
11.            Sayable sayable = methodReference::saySomething;
12.        // Calling interface method
13.            sayable.say();
14.            // Referring non-static method using anonymous object
15.            Sayable sayable2 = new InstanceMethodReference()::saySomething; // You can use anon
    ymous object also
16.            // Calling interface method
17.            sayable2.say();
18.    }
19. }
```

Output:

```
Hello, this is non-static method.
Hello, this is non-static method.
```

3) Reference to a Constructor

You can refer a constructor by using the new keyword. Here, we are referring constructor with the help of functional interface.

Syntax

ClassName::**new**

Example

```
1. interface Messageable{
2.     Message getMessage(String msg);
3. }
4. class Message{
5.     Message(String msg){
6.         System.out.print(msg);
7.     }
8. }
9. public class ConstructorReference {
10.     public static void main(String[] args) {
11.         Messageable hello = Message::new;
12.         hello.getMessage("Hello");
13.     }
14. }
```

Output:

```
Hello
```

Java Type and Repeating Annotations

Java Type Annotations

Java 8 has included two new features repeating and type annotations in its prior annotations topic. In early Java versions, you can apply annotations only to declarations. After releasing of Java SE 8 , annotations can be applied to any type use. It means that annotations can be used anywhere you use a type. For example, if you want to avoid NullPointerException in your code, you can declare a string variable like this:

@NonNull String str;

Following are the examples of type annotations:

@NonNull List<String>
List<@NonNull String> str
Arrays<@NonNegative Integer> sort
@Encrypted File file
@Open Connection connection
void divideInteger(**int** a, **int** b) **throws** @ZeroDivisor ArithmeticException

Java Repeating Annotations

In Java 8 release, Java allows you to repeating annotations in your source code. It is helpful when you want to reuse annotation for the same class. You can repeat an annotation anywhere that you would use a standard annotation.

For compatibility reasons, repeating annotations are stored in a container annotation that is automatically generated by the Java compiler. In order for the compiler to do this, two declarations are required in your code.

1. Declare a repeatable annotation type
2. Declare the containing annotation type

---

1) Declare a repeatable annotation type

Declaring of repeatable annotation type must be marked with the @Repeatable meta-annotation. In the following example, we have defined a custom @Game repeatable annotation type.

```
1. @Repeatable(Games.class)
2. @interfaceGame{
3.     String name();
4.     String day();
5. }
```

The value of the @Repeatable meta-annotation, in parentheses, is the type of the container annotation that the Java compiler generates to store repeating annotations. In the following example, the containing annotation type is Games. So, repeating @Game annotations is stored in an @Games annotation.

---

2) Declare the containing annotation type

Containing annotation type must have a value element with an array type. The component type of the array type must be the repeatable annotation type. In the following example, we are declaring Games containing annotation type:

```
1. @interfaceGames{
2.     Game[] value();
3. }
```

Note - Compiler will throw a compile-time error, if you apply the same annotation to a declaration without first declaring it as repeatable.

Java Repeating Annotations Example

```
1. // Importing required packages for repeating annotation
2. import java.lang.annotation.Repeatable;
3. import java.lang.annotation.Retention;
4. import java.lang.annotation.RetentionPolicy;
5. // Declaring repeatable annotation type
6. @Repeatable(Games.class)
7. @interfaceGame{
8.     String name();
9.     String day();
10. }
11. // Declaring container for repeatable annotation type
12. @Retention(RetentionPolicy.RUNTIME)
13. @interfaceGames{
14.     Game[] value();
15. }
16. // Repeating annotation
17. @Game(name = "Cricket",  day = "Sunday")
18. @Game(name = "Hockey",   day = "Friday")
19. @Game(name = "Football", day = "Saturday")
20. public class RepeatingAnnotationsExample {
21.     public static void main(String[] args) {
22.         // Getting annotation by type into an array
23.         Game[] game = RepeatingAnnotationsExample.class.getAnnotationsByType(Game.class);
```

```
24.         for (Gamegame2 : game) {    // Iterating values
25.             System.out.println(game2.name()+" on "+game2.day());
26.         }
27.     }
28. }
```

OUTPUT:

```
Cricket on Sunday
Hockey on Friday
Football on Saturday
```

Method Parameter Reflection

Java provides a new feature in which you can get the names of formal parameters of any method or constructor. The java.lang.reflect package contains all the required classes like Method and Parameter to work with parameter reflection.

Method class

It provides information about single method on a class or interface. The reflected method may be a class method or an instance method.

Method Class methods

Method	Description
public AnnotatedType getAnnotatedReturnType()	It returns an AnnotatedType object that represents the use of a type to specify the return type of the method/constructor.
public Class<?> getDeclaringClass()	It returns the Class object representing the class or interface that declares the executable represented by this object.
public Object getDefaultValue()	It returns the default value for the annotation member represented by this Method instance.

public Class<?>[] getExceptionTypes()	It returns an array of Class objects that represent the types of exceptions declared to be thrown by the underlying executable represented by this object.
public int getModifiers()	It returns the Java language modifiers for the executable represented by this object.
public String getName()	It returns the name of the method represented by this Method object as a String.
public int getParameterCount()	It returns the number of formal parameters for the executable represented by this object.
public Class<?> getReturnType()	It returns a Class object that represents the formal Return type of the method represented by this Method object.
public int hashCode()	It returns a hashcode for this Method. The hashcode is computed as the exclusive-or of the hashcodes for the underlying method's declaring class name and the method's name.
public boolean isBridge()	It returns true if this method is a bridge method. otherwise returns false.
public String toGenericString()	It returns a string describing this Method, including type parameters.
public String toString()	It returns a string.

Parameter class

Parameter class provides information about method parameters, including its name and modifiers. It also provides an alternate means of obtaining attributes for the parameter.

Parameter Methods

Methods	Description
public boolean equals(Object obj)	It compares based on the executable and the index.
public AnnotatedType getAnnotatedType()	It returns an AnnotatedType object that represents the use of a type to specify the type of the formal parameter represented by this Parameter.
public Annotation[] getAnnotations()	It returns annotations that are present on this element. If there are no annotations present on this element, the return value is an array of length 0.
public int getModifiers()	It returns the modifier flags for the parameter represented by this Parameter object.
public Type getParameterizedType()	It returns a Type object that identifies the parameterized type for the parameter represented by this Parameter object.
public Class<?> getType()	It returns a Class object that identifies the declared type for the parameter represented by this Parameter object.
public boolean isImplicit()	It returns true if this parameter is implicitly declared in source code. Otherwise, returns false.
public String toString()	It returns a string describing this parameter. The format is The modifiers for the parameter, if any, in canonical order as recommended by The Java? Language Specification.

Java Method Parameter Reflection Example

File: Calculate.java

```
1. public class Calculate {
2.     int add(int a, int b){
3.         return (a+b);
4.     }
5.     int mul(int a, int b){
6.         return (b*a);
```

```
7.      }
8.  }
```

How to set path in Java

1. How to set the path of JDK in Windows OS
1. Setting Temporary Path of JDK
2. Setting Permanent Path of JDK
2. How to set the path of JDK in Linux OS

The path is required to be set for using tools such as javac, java, etc.

If you are saving the Java source file inside the JDK/bin directory, the path is not required to be set because all the tools will be available in the current directory.

However, if you have your Java file outside the JDK/bin folder, it is necessary to set the path of JDK.

There are two ways to set the path in Java:

1. Temporary
2. Permanent

1) How to set the Temporary Path of JDK in Windows

To set the temporary path of JDK, you need to follow the following steps:

- Open the command prompt
- Copy the path of the JDK/bin directory
- Write in command prompt: set path=copied_path

For Example:

```
set path=C:\Program Files\Java\jdk1.6.0_23\bin
```

2) How to set Permanent Path of JDK in Windows

For setting the permanent path of JDK, you need to follow these steps:

- Go to MyComputer properties -> advanced tab -> environment variables -> new tab of user variable -> write path in variable name -> write path of bin folder in variable value -> ok -> ok -> ok

For Example:

1) Go to MyComputer properties

2) Click on the advanced tab

3) Click on environment variables

4) Click on the new tab of user variables

5) Write the path in the variable name

6) Copy the path of bin folder

7) Paste path of bin folder in the variable value

8) Click on ok button

9) Click on ok button

Now your permanent path is set. You can now execute any program of java from any drive.

Interpreter	***Compiler***
Translates program one statement at a time.	Scans the entire program and translates it as a whole into machine code.
It takes less amount of time to analyze the source code but the overall execution time is slower.	It takes large amount of time to analyze the source code but the overall execution time is comparatively faster.
No intermediate object code is generated, hence are memory efficient.	Generates intermediate object code which further requires linking, hence requires more memory.
Continues translating the program until the first error is met, in which case it stops. Hence debugging is easy.	It generates the error message only after scanning the whole program. Hence debugging is comparatively hard.
Programming language like Python, Ruby use interpreters.	Programming language like C, C++ use compilers.

Java Program

```
1. class Simple{
2.    public static void main(String args[]){
3.     System.out.println("Hello Java");
4.    }
5. }
```

save this file as Simple.java

To compile: javac Simple.java

To execute: java Simple

Output: Hello Java

Java Applications

There are mainly 4 types of applications that can be created using Java programming:

1) Standalone Application

Standalone applications are also known as desktop applications or window-based applications. These are traditional software that we need to install on every machine. Examples of standalone application are Media player, antivirus, etc. AWT and Swing are used in Java for creating standalone applications.

2) Web Application

An application that runs on the server side and creates a dynamic page is called a web application. Currently, Servlet, JSP,Struts, Spring, Hibernate, JSF, etc. technologies are used for creating web applications in Java.

3) Enterprise Application

An application that is distributed in nature, such as banking applications, etc. is called enterprise application. It has advantages of the high-level security, load balancing, and clustering. In Java, EJB is used for creating enterprise applications.

4) Mobile Application

An application which is created for mobile devices is called a mobile application. Currently, Android and Java ME are used for creating mobile applications.

public

This is the access modifier of the main method. It has to be public so that java runtime can execute this method. Remember that if you make any method non-public then it's not allowed to be executed by any program, there are some access restrictions applied. So it means that the main method has to be public. Let's see what happens if we define the main method as non-public.

```
public class Test {
```

```
static void main(String[] args){

        System.out.println("Hello World");
}}
$ javac Test.java
$ java Test
Error: Main method not found in class Test, please define the main method as:
  public static void main(String[] args)
or a JavaFX application class must extend javafx.application.Application
$
```

static

When java runtime starts, there is no object of the class present. That's why the main method has to be static so that JVM can load the class into memory and call the main method. If the main method won't be static, JVM would not be able to call it because there is no object of the class is present. Let's see what happens when we remove static from java main method.

```
public class Test {

public void main(String[] args){

        System.out.println("Hello World");

}
}

$ javac Test.java
$ java Test
Error: Main method is not static in class Test, please define the main method as:
  public static void main(String[] args)
$
```

void

Java programming mandates that every method provide the return type. Java main method doesn't return anything, that's why it's return type is void. This has been done to keep things simple because once the main method is finished executing, java program terminates. So there is no point in returning anything, there is nothing that can be done for the returned object by JVM. If we try to return something from the main method, it will give compilation error as an unexpected return value. For example, if we have the main method like below.

```
public class Test {

public static void main(String[] args){

        return 0;
}
}
```

We get below error when above program is compiled.

```
$ javac Test.java
Test.java:5: error: incompatible types: unexpected return value
        return 0;
            ^
1 error
$
```

main

This is the name of java main method. It's fixed and when we start a java program, it looks for the main method. For example, if we have a class like below.

```
public class Test {

public static void mymain(String[] args){
```

```
        System.out.println("Hello World");
}
}
```

And we try to run this program, it will throw an error that the main method is not found.

```
$ javac Test.java
$ java Test
Error: Main method not found in class Test, please define the main method as:
   public static void main(String[] args)
or a JavaFX application class must extend javafx.application.Application
$
```

String[] args

Java main method accepts a single argument of type String array. This is also called as java command line arguments. Let's have a look at the example of using java command line arguments.

```
public class Test {

public static void main(String[] args){

   for(String s : args){
        System.out.println(s);
   }

   }
}
```

Above is a simple program where we are printing the command line arguments. Let's see how to pass command line arguments when executing above program.

```
$ javac Test.java
$ java Test 1 2 3
1
2
3
$ java Test "Hello World" "Pankaj Kumar"
Hello World
Pankaj Kumar
$ java Test
$
```

Java Statements

Statements are everything that make up a complete unit of execution. For example,

```
int score = 9*5;
```

Here, 9*5 is an expression that returns 45, and int score = 9*5; is a statement.

Expressions are part of statements.

Expression statements

Some expressions can be made into statement by terminating the expression with a ;. These are known as expression statements. For example:

```
number = 10;
```

Here, number = 10 is an expression where as number = 10; is a statement that compiler can execute.

```
++number;
```

Here, ++number is an expression where as ++number; is a statement.

Declaration Statements

Declaration statements declares variables. For example,

```
Double tax = 9.5;
```

The statement above declares a variable tax which is initialized to 9.5.

Also, there are control flow statements that are used in decision making and looping in Java. You will learn about control flow statements in later chapters.

Java Keywords

Keywords are predefined, reserved words used in Java programming that have special meanings to the compiler. For example:

```
int score;
```

Here, int is a keyword. It indicates that the variable score is of integer type (32-bit signed two's complement integer).

You cannot use keywords like int, for, class etc as variable name (or identifiers) as they are part of the Java programming language syntax. Here's the complete list of all keywords in Java programming.

Java Keywords List

abstract	assert	boolean	break	byte

case	catch	char	class	const
continue	default	do	double	else
enum	extends	final	finally	float
for	goto	if	implements	import
instanceof	int	interface	long	native
new	package	private	protected	public
return	short	static	strictfp	super
switch	synchronized	this	throw	throws
transient	try	void	volatile	while

Beside these keywords, you cannot also use true, false and null as identifiers as they are literals.

Java identifiers

Identifiers are the name given to variables, classes, methods etc. Consider the above code;

```
int score;
```

Here, score is a variable (an identifier). You cannot use keywords as variable name. It's because keywords have predefined meaning. For example,

```
int float;
```

The above code is wrong. It's because float is a keyword and cannot be used as a variable name.

To learn more about variables, visit Java variables.

Rules for Naming an Identifier

- Identifier cannot be a keyword.
- Identifiers are case-sensitive.
- It can have a sequence of letters and digits. However, it must begin with a letter, $ or_. The first letter of an identifier cannot be a digit.
- It's convention to start an identifier with a letter rather and $ or _.
- Whitespaces are not allowed.
- Similarly, you cannot use symbols such as @, #, and so on.

Here are some valid identifiers:

- score
- level
- highestScore
- number1
- convertToString

Here are some invalid identifiers:

- class
- float
- 1number
- highest Score
- @pple

Java Comments

The java comments are statements that are not executed by the compiler and interpreter. The comments can be used to provide information or explanation about the variable, method, class or any statement. It can also be used to hide program code for specific time.

Types of Java Comments

There are 3 types of comments in java.

1. **Single Line Comment**
2. **Multi Line Comment**
3. **Documentation Comment**

1) Java Single Line Comment

The single line comment is used to comment only one line.

Syntax:

```
1. //This is single line comment
```

Example:

```
1. public class CommentExample1 {
2. public static void main(String[] args) {
3.     int i=10;//Here, i is a variable
4.     System.out.println(i);
5. }
6. }
```

Output:

```
10
```

2) Java Multi Line Comment

The multi line comment is used to comment multiple lines of code.

Syntax:

```
1. /*
2. This
3. is
4. multi line
5. comment
6. */
```

Example:

```
1. public class CommentExample2 {
2. public static void main(String[] args) {
3. /* Let's declare and
4.  print variable in java. */
5.     int i=10;
6.     System.out.println(i);
7. }
8. }
```

Output:

```
10
```

3) Java Documentation Comment

The documentation comment is used to create documentation API. To create documentation API, you need to use javadoc tool.

Syntax:

```
1. /**
2. This
3. is
4. documentation
5. comment
6. */
```

Example:

```
1. /** The Calculator class provides methods to get addition and subtraction of given 2 numbers.*/
2. public class Calculator {
3. /** The add() method returns addition of given numbers.*/
4. public static int add(int a, int b){return a+b;}
5. /** The sub() method returns subtraction of given numbers.*/
6. public static int sub(int a, int b){return a-b;}
7. }
```

Java - Variable Types

A variable provides us with named storage that our programs can manipulate. Each variable in Java has a specific type, which determines the size and layout of the variable's memory; the range of values that can be stored within that memory; and the set of operations that can be applied to the variable.

You must declare all variables before they can be used. Following is the basic form of a variable declaration −

```
data type variable [ = value][, variable [ = value] ...] ;
```

Here data type is one of Java's datatypes and variable is the name of the variable. To declare more than one variable of the specified type, you can use a comma-separated list.

Following are valid examples of variable declaration and initialization in Java −

Example

```
int a, b, c;         // Declares three ints, a, b, and c.
int a = 10, b = 10;  // Example of initialization
byte B = 22;         // initializes a byte type variable B.
double pi = 3.14159; // declares and assigns a value of PI.
char a = 'a';        // the char variable a iis initialized with value 'a'
```

This chapter will explain various variable types available in Java Language. There are three kinds of variables in Java −

- Local variables
- Instance variables
- Class/Static 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.

Live Demo

```
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 −

Output

```
Puppy age is: 7
```

Example

Following example uses age without initializing it, so it would give an error at the time of compilation.

Live Demo

```
public class Test {

  public void pupAge() {

    int age;

    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 error while compiling it −

Output

```
Test.java:4:variable number might not have been initialized
age = age + 7;
       ^
1 error
```

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 −

Output

```
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 nameClassName.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 −

Output

```
Development average salary:1000
```

Data Types in Java

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

1. **Primitive data types:** The primitive data types include Integer, Character, Boolean, and Floating Point.

2. **Non-primitive data types:** The non-primitive data types include Classes, Interfaces, and Arrays.

Java Primitive Data Types
In Java language, primitive data types are the building blocks of data manipulation. These are the most basic data types available in Java language. Java is a statically-typed programming language. It means, all variables must be declared before its use. That is why we need to declare variable's type and name.
There are 8 types of primitive data types:

o boolean data type

o byte data type

o char data type

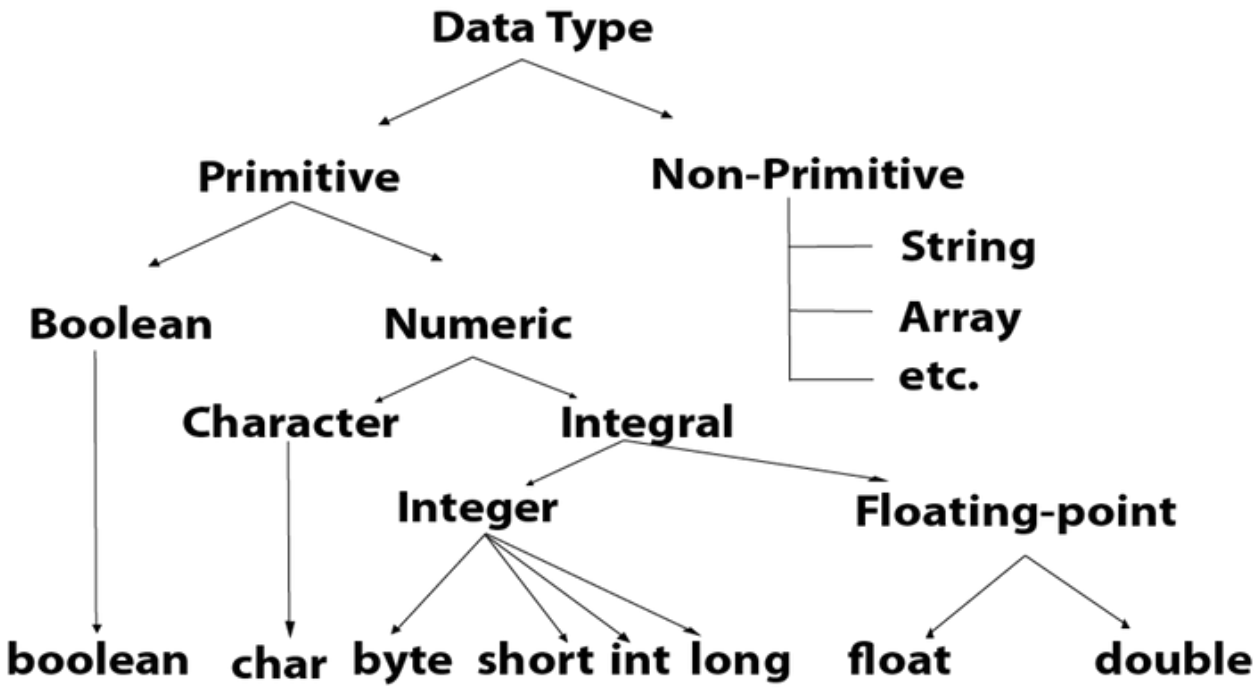
o short data type

o int data type

o long data type

o float data type

o double data type



Data Type	Default Value	Default size
boolean	false	1 bit
char	'\u0000'	2 byte
byte	0	1 byte
short	0	2 byte
int	0	4 byte
long	0L	8 byte
float	0.0f	4 byte
double	0.0d	8 byte

Boolean Data Type

The Boolean data type is used to store only two possible values: true and false. This data type is used for simple flags that track true/false conditions.

The Boolean data type specifies one bit of information, but its "size" can't be defined precisely.

Example: Boolean one = false

Byte Data Type

The byte data type is an example of primitive data type. It isan 8-bit signed two's complement integer. Its value-range lies between -128 to 127 (inclusive). Its minimum value is -128 and maximum value is 127. Its default value is 0.

The byte data type is used to save memory in large arrays where the memory savings is most required. It saves space because a byte is 4 times smaller than an integer. It can also be used in place of "int" data type.

Example: byte a = 10, byte b = -20

Short Data Type

The short data type is a 16-bit signed two's complement integer. Its value-range lies between -32,768 to 32,767 (inclusive). Its minimum value is -32,768 and maximum value is 32,767. Its default value is 0.

The short data type can also be used to save memory just like byte data type. A short data type is 2 times smaller than an integer.

Example: short s = 10000, short r = -5000

Int Data Type

The int data type is a 32-bit signed two's complement integer. Its value-range lies between -2,147,483,648 (-2^31) to 2,147,483,647 (2^31 -1) (inclusive). Its minimum value is -2,147,483,648and maximum value is 2,147,483,647. Its default value is 0.

The int data type is generally used as a default data type for integral values unless if there is no problem about memory.

Example: int a = 100000, int b = -200000

Long Data Type

The long data type is a 64-bit two's complement integer. Its value-range lies between -9,223,372,036,854,775,808(-2^63) to 9,223,372,036,854,775,807(2^63 -1)(inclusive). Its minimum value is - 9,223,372,036,854,775,808and maximum value is 9,223,372,036,854,775,807. Its default value is 0. The long data type is used when you need a range of values more than those provided by int.

Example: long a = 100000L, long b = -200000L

Float Data Type

The float data type is a single-precision 32-bit IEEE 754 floating point.Its value range is unlimited. It is recommended to use a float (instead of double) if you need to save memory in large arrays of floating point numbers. The float data type should never be used for precise values, such as currency. It is generally used as the default data type for decimal values. Its default value is 0.0d.

Example: float f1 = 234.5f

Double Data Type

The double data type is a double-precision 64-bit IEEE 754 floating point. Its value range is unlimited. The double data type is generally used for decimal values just like float. The double data type also should never be used for precise values, such as currency.Its default value is 0.0d.

Example: double d1 = 12.3

Char Data Type

The char data type is a single 16-bit Unicode character. Its value-range lies between '\u0000' (or 0) to '\uffff' (or 65,535 inclusive).The char data type is used to store characters.

Example: char letterA = 'A'

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 −

Example

```
"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
\\	backslash
\ddd	Octal character (ddd)
\uxxxx	Hexadecimal UNICODE character (xxxx)

Java - Basic Operators

Java provides a rich set of operators to manipulate variables. We can divide all the Java operators into the following groups −

- Arithmetic Operators
- Relational Operators
- Bitwise Operators
- Logical Operators
- Assignment Operators
- Misc Operators

The 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 −

Assume integer variable A holds 10 and variable B holds 20, then −

Show Examples

Operator	Description	Example

+ (Addition)	Adds values on either side of the operator.	A + B will give 30
- (Subtraction)	Subtracts right-hand operand from left-hand operand.	A - B will give -10
* (Multiplication)	Multiplies values on either side of the operator.	A * B will give 200
/ (Division)	Divides left-hand operand by right-hand operand.	B / A will give 2
% (Modulus)	Divides left-hand operand by right-hand operand and returns remainder.	B % A will give 0
++ (Increment)	Increases the value of operand by 1.	B++ gives 21
-- (Decrement)	Decreases the value of operand by 1.	B-- gives 19

The Relational Operators

There are following relational operators supported by Java language.

Assume variable A holds 10 and variable B holds 20, then −

Show Examples

Operator	Description	Example
== (equal to)	Checks if the values of two operands are equal or not, if yes then condition becomes true.	(A == B) is not true.
!= (not equal to)	Checks if the values of two operands are equal or not, if values are not equal then condition becomes true.	(A != B) is true.

> (greater than)	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	(A > B) is not true.
< (less than)	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	(A < B) is true.
>= (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.	(A >= B) is not true.
<= (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.	(A <= B) is true.

The Bitwise Operators

Java defines several bitwise operators, which can be applied to the integer types, long, int, short, char, and byte.

Bitwise operator works on bits and performs bit-by-bit operation. Assume if a = 60 and b = 13; now in binary format they will be as follows −

a = 0011 1100

b = 0000 1101

---

a&b = 0000 1100

a|b = 0011 1101

a^b = 0011 0001

~a = 1100 0011

The following table lists the bitwise operators −

Assume integer variable A holds 60 and variable B holds 13 then −

Show Examples

Operator	Description	Example

& (bitwise and)	Binary AND Operator copies a bit to the result if it exists in both operands.	(A & B) will give 12 which is 0000 1100
| (bitwise or)	Binary OR Operator copies a bit if it exists in either operand.	(A | B) will give 61 which is 0011 1101
^ (bitwise XOR)	Binary XOR Operator copies the bit if it is set in one operand but not both.	(A ^ B) will give 49 which is 0011 0001
~ (bitwise compliment)	Binary Ones Complement Operator is unary and has the effect of 'flipping' bits.	(~A) will give -61 which is 1100 0011 in 2's complement form due to a signed binary number.
<< (left shift)	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.	A << 2 will give 240 which is 1111 0000
>> (right shift)	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.	A >> 2 will give 15 which is 1111
>>> (zero fill right shift)	Shift right zero fill operator. The left operands value is moved right by the number of bits specified by the right operand and shifted values are filled up with zeros.	A >>>2 will give 15 which is 0000 1111

The Logical Operators

The following table lists the logical operators −

Assume Boolean variables A holds true and variable B holds false, then −

Show Examples

Operator	Description	Example
&& (logical and)	Called Logical AND operator. If both the operands	(A && B) is false

	are non-zero, then the condition becomes true.	
|| (logical or)	Called Logical OR Operator. If any of the two operands are non-zero, then the condition becomes true.	(A || B) is true
! (logical not)	Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true then Logical NOT operator will make false.	!(A && B) is true

The Assignment Operators

Following are the assignment operators supported by Java language −

Show Examples

Operator	Description	Example
=	Simple assignment operator. Assigns values from right side operands to left side operand.	C = A + B will assign value of A + B into C
+=	Add AND assignment operator. It adds right operand to the left operand and assign the result to left operand.	C += A is equivalent to C = C + A
-=	Subtract AND assignment operator. It subtracts right operand from the left operand and assign the result to left operand.	C -= A is equivalent to C = C – A
*=	Multiply AND assignment operator. It multiplies right operand with the left operand and assign the result to left operand.	C *= A is equivalent to C = C * A

/=	Divide AND assignment operator. It divides left operand with the right operand and assign the result to left operand.	C /= A is equivalent to C = C / A
%=	Modulus AND assignment operator. It takes modulus using two operands and assign the result to left operand.	C %= A is equivalent to C = C % A
<<=	Left shift AND assignment operator.	C <<= 2 is same as C = C << 2
>>=	Right shift AND assignment operator.	C >>= 2 is same as C = C >> 2
&=	Bitwise AND assignment operator.	C &= 2 is same as C = C & 2
^=	bitwise exclusive OR and assignment operator.	C ^= 2 is same as C = C ^ 2
|=	bitwise inclusive OR and assignment operator.	C |= 2 is same as C = C | 2

Miscellaneous Operators

There are few other operators supported by Java Language.

Conditional Operator (? :)

Conditional operator is also known as the ternary operator. This operator consists of three operands and is used to evaluate Boolean expressions. The goal of the operator is to decide, which value should be assigned to the variable. The operator is written as −

```
variable x = (expression) ? value if true : value if false
```

Following is an example −

Example

```
public class Test {


  public static void main(String args[]) {
    int a, b;
    a = 10;
    b = (a == 1) ? 20: 30;
    System.out.println( "Value of b is : " +  b );


    b = (a == 10) ? 20: 30;
    System.out.println( "Value of b is : " + b );
  }
}
```

This will produce the following result −

Output

```
Value of b is : 30
Value of b is : 20
```

instanceof Operator

This operator is used only for object reference variables. The operator checks whether the object is of a particular type (class type or interface type). instanceof operator is written as −

```
( Object reference variable ) instanceof  (class/interface type)
```

If the object referred by the variable on the left side of the operator passes the IS-A check for the class/interface type on the right side, then the result will be true. Following is an example −

Example

```
public class Test {

  public static void main(String args[]) {

    String name = "James";

    // following will return true since name is type of String
    boolean result = name instanceof String;
    System.out.println( result );
  }
}
```

This will produce the following result −

Output

```
true
```

This operator will still return true, if the object being compared is the assignment compatible with the type on the right. Following is one more example −

Example

```
class Vehicle {}
```

```
public class Car extends Vehicle {


  public static void main(String args[]) {


    Vehicle a = new Car();
    boolean result =  a instanceof Car;
    System.out.println( result );
  }
}
```

This will produce the following result −

Output

```
true
```

Precedence of Java Operators

Operator precedence determines the grouping of terms in an expression. This affects how an expression is evaluated. Certain operators have higher precedence than others; for example, the multiplication operator has higher precedence than the addition operator −

For example, x = 7 + 3 * 2; here x is assigned 13, not 20 because operator * has higher precedence than +, so it first gets multiplied with 3 * 2 and then adds into 7.

Here, operators with the highest precedence appear at the top of the table, those with the lowest appear at the bottom. Within an expression, higher precedence operators will be evaluated first.

Category	Operator	Associativity
Postfix	expression++ expression--	Left to right
Unary	++expression –-expression +expression –expression ~ !	Right to left
Multiplicative	* / %	Left to right

Additive	+ -	Left to right
Shift	<< >> >>>	Left to right
Relational	< > <= >= instanceof	Left to right
Equality	== !=	Left to right
Bitwise AND	&	Left to right
Bitwise XOR	^	Left to right
Bitwise OR	|	Left to right
Logical AND	&&	Left to right
Logical OR	||	Left to right
Conditional	?:	Right to left
Assignment	= += -= *= /= %= ^= |= <<= >>= >>>=	Right to left

Strings Class

Strings, which are widely used in Java programming, are a sequence of characters. In Java programming language, strings are treated as objects.

The Java platform provides the String class to create and manipulate strings.

Creating Strings

The most direct way to create a string is to write −

```
String greeting = "Hello world!";
```

Whenever it encounters a string literal in your code, the compiler creates a String object with its value in this case, "Hello world!'.

As with any other object, you can create String objects by using the new keyword and a constructor. The String class has 11 constructors that allow you to provide the initial value of the string using different sources, such as an array of characters.

Example
Live Demo

```
public class StringDemo {


  public static void main(String args[]) {
    char[] helloArray = { 'h', 'e', 'l', 'l', 'o', '.' };
    String helloString = new String(helloArray);
    System.out.println( helloString );
  }
}
```

This will produce the following result −

Output

```
hello.
```

Note − The String class is immutable, so that once it is created a String object cannot be changed. If there is a necessity to make a lot of modifications to Strings of characters, then you should use String Buffer & String BuilderClasses.

Note :- For More Programs , Operators , String Class and Methods (String Buffer) With its Examples Refer Class Note Book.

-----For Your Rough Work----