Welcome you all

JAVA PROGRAMMING

DAY : 1&2

D.Sakthivel
Assistant Professor & Trainer,
KGiSL Micro College
KGiSL Campus, Coimbatore – 641 035.

Day 1&2

- Introduction
- History
- Features
- Java Installation and Path setting
- Java Data types
- Variables
- Operators
- Decision Branching
- Decision Looping
- Arrays
- String

KGiSL MicroCollege

Java Programming

-Java is a **programming language** and a **platform independent language**.

-Java is a high level, robust, object-oriented and secure programming language.

-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 r a.

Java Programming

Java is one of the most popular programming languages.

“Write once, run anywhere”

Java's slogan is "**Write once, run anywhere**".

This means that the same Java code can run on different platforms, including mobile, desktop and other portable systems.

Platform: Any hardware or software environment in which a program runs, is known as a platform. Since Java has a runtime e l API, it is called a p

Application

- Desktop Applications such as acrobat reader, media player, antivirus, etc.
- Web Applications such as irctc.co.in, javatpoint.com, etc.
- Enterprise Applications such as banking applications.
- Mobile
- Embedded System
- Smart Card
- Robotics
- Games, etc.

Types of 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.

Servlet, JSP, Struts, Spring, Hibernate, JSF, etc. technologies are used for creating web applications in Java.

Types of Java Applications

3) Enterprise Application

An application that is distributed in nature, such as banking applications, etc. is called an enterprise application. It has advantages like 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.

Top Mobile & Web Applications of Java in Real World

- Spotify (Music Streaming App) ...
- Twitter (Social Media App) ...
- Opera Mini (Web Browser) ...
- Nimbuzz Messenger (Instant Messaging App) ...
- CashApp (Mobile Payment Service) ...
- ThinkFree Office (Desktop-based App) ...
- Signal (Encrypted Messaging Services) ...
- Murex (Trading System)

- **Google & Android OS**
- **NETFLIX (JAVA & Python)**
- **LinkedIn**
- **Uber**
- **Amazon**

KGiSL MicroCollege

Java Platforms / Editions

There are 4 platforms or editions of Java:

1) Java SE (Java Standard Edition)

- It is a Java programming platform.
- It includes Java programming APIs such as java.lang, java.io, java.net, java.util, java.sql, java.math etc.
- It includes core topics like OOPs, String, Regex, Exception, Inner classes, Multithreading, I/O Stream, Networking, AWT, Swing, Reflection, Collection, etc.

2) Java EE (Java Enterprise Edition)

- It is an enterprise platform that is mainly used to develop web and enterprise applications.
- It is built on top of the Java SE platform. It includes topics like Servlet, JSP, Web Services, EJB, JPA, etc.

Java Platforms / Editions

3) Java ME (Java Micro Edition)

- It is a micro platform that is dedicated to mobile applications.

4) JavaFX

- It is used to develop rich internet applications. It uses a lightweight user interface API.

History of Java

- Java was originally designed for interactive television, initiated this project to develop a language for digital devices such as set-top boxes, televisions, etc.
- Currently, Java is used in internet programming, mobile devices, games, e-business solutions, etc.
- **James Gosling, Mike Sheridan**, and **Patrick Naughton** initiated the Java language project in June 1991.
- The small team of sun engineers called **Green Team**.

History of Java

Why Java was named as "Oak"?

Why Oak? Oak is a symbol of strength and chosen as a national tree of many countries like the U.S.A., France, Germany, Romania, etc.

In 1995, Oak was renamed as **"Java"** because it was already a trademark by Oak Technologies.

Why Java Programming named "Java"?

- Java is an island in Indonesia where the first coffee was produced (called Java coffee). It is a kind of espresso bean. Java name was chosen by James Gosling while having a cup of coffee nearby his office.
- Notice that Java is just a name, not an acronym.
- Initially developed by James Gosling at Sun Microsystems (which is now a subsidiary of Oracle Corporation) and released in 1995.
- n January 23, 1996. Java Version

Many java versions have been released till now. The current stable release of Java is Java SE 10.

JDK Alpha and Beta (1995)
JDK 1.0 (23rd Jan 1996)
JDK 1.1 (19th Feb 1997)
J2SE 1.2 (8th Dec 1998)
J2SE 1.3 (8th May 2000)
J2SE 1.4 (6th Feb 2002)
J2SE 5.0 (30th Sep 2004)
Java SE 6 (11th Dec 2006)
Java SE 7 (28th July 2011)
Java SE 8 (18th Mar 2014)
Java SE 9 (21st Sep 2017)
Java SE 10 (20th Mar 2018)
Java SE 11 (September 2018)
Java SE 12 (March 2019)
Java SE 13 (September 2019)
Java SE 14 (Mar 2020)
Java SE 15 (September 2020)
Java SE 16 (Mar 2021)
Java SE 17 (September 2021)
Java SE 18 (to be released by March 2022)

Features of Java

A list of the most important features of the Java language is given below.

- Simple
- Object-Oriented
- Portable
- Platform independent
- Secured
- Robust
- Architecture neutral
- Interpreted
- High Performance
- Multithreaded
- Distributed
- Dynamic

Features of Java

- **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.

Features of Java

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 behavior.
- Object-oriented programming (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

Platform Independent

Java is platform independent because it is different from other languages like C, C++, etc. which are 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.

There are two types of platforms software-based and hardware-based. Java provides a software-based platform.

The Java platform differs from most other platforms in the sense that it is a software-based platform that runs on top of other hardware-based platforms. It has two components:

1. Runtime Environment

2. API(Application Programming Interface)

Java code can be executed on multiple platforms, for example, Windows, Linux, Sun Solaris, Mac/OS, etc.

- **Java code is compiled by the compiler and converted into bytecode.**
- **This bytecode is a platform-independent code because it can be run on multiple platforms, i.e., W and Run Anywhere (WORA).**

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 machine sandbox**

Classloader: Class loader in Java is a part of the Java Runtime Environment (JRE) which is used to load Java classes into the Java Virtual Machine dynamically.

It adds security by separating the package for the classes of the local file system from those that are imported from network sources.

Secured

Bytecode Verifier: It checks the code fragments for illegal code that can violate access rights to objects.

Security Manager: It determines what resources a class can access such as reading and writing to the local disk.

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.

Architecture-neutral

- Java is architecture neutral because there are no implementation dependent features, for example, the size of primitive types is fixed.
- In C programming, int data type occupies 2 bytes of memory for 32-bit architecture and 4 bytes of memory for 64-bit architecture.
- However, it occupies 4 bytes of memory for both 32 and 64-bit architectures in Java.

Portable

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

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.

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.

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.

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++.
- Java supports dynamic compilation and automatic memory management (garbage collection).

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**

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

Setting Java Path in Linux OS

Setting path in Linux OS is the same as setting the path in the Windows OS. But, here we use the export tool rather than set.

Let's see how to set path in Linux OS:

export PATH=$PATH:/home/jdk1.6.01/bin/

Here, we have installed the JDK in the home directory under Root (/home).

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.
- JVMs are available for many hardware and software platforms There are three notions of the JVM: *specification*, *implementation*. and *instance*.

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.
- The implementation of JVM is also actively released by other companies besides Sun Micro Systems.

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

.

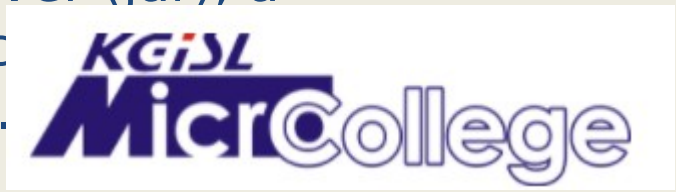
It contains JRE + development tools.

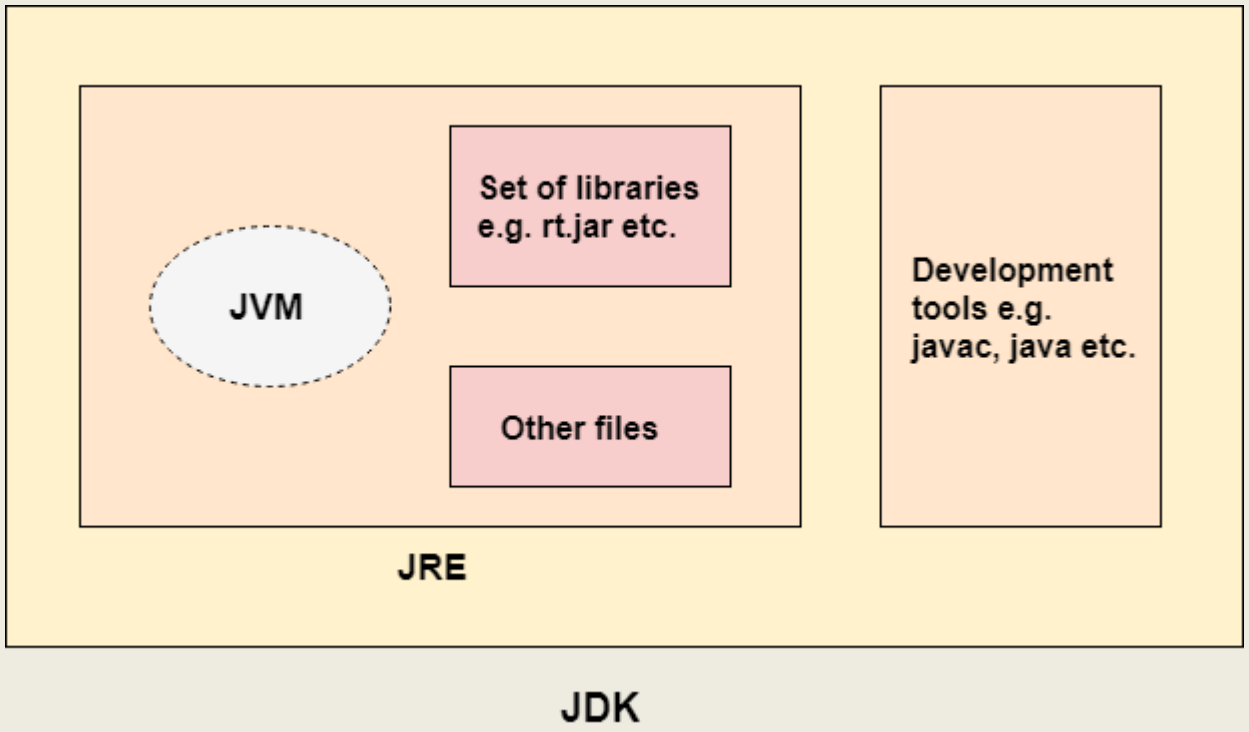
JDK is an implementation of any one of the below given Java Platforms released by Oracle Corporation:

- **Standard Edition Java Platform**
- **Enterprise Edition Java Platform**

Micro Edition Java Platform

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 development of a Java Application.




Set of libraries
e.g. rt.jar etc.
JVM
Other files
Development
tools e.g.
javac, java etc.
JRE
JDK

Java ?

Programming

To get started :

1. java jdk
2. IDE
   (0r) online compilers

INSIDE JAVA

Package

class

methods

JAVA PROGRAM

.java file Structure ?

```
package com.Demo;

public class Main {

public static void main(String[] args){
   // write your code here

    }
}
```

How java code gets executed

.java file -> java compiler -> .class file

Actually the fact is Java platform independent then but JVM (Java Virtual Machine) is platform dependent.
There are two tools which use for compiling and running Java programs

javac -It is a compiler which converts Java source code to byte code that is a .class file. This byte code is standard for all platforms, machines or operating systems.

Java – It is an interpreter. This interprets the .class file based on a particular platform and executes them.

Jvm -Java virtual machine comes into play. Jvm for windows will be different from Jvm for Solaris or Linux. But all the Jvm take the same byte code and executes them in that platform.

Java editions

SE	EE	ME	java card
(using now)	(company)	(mobiles)	(smart cards)

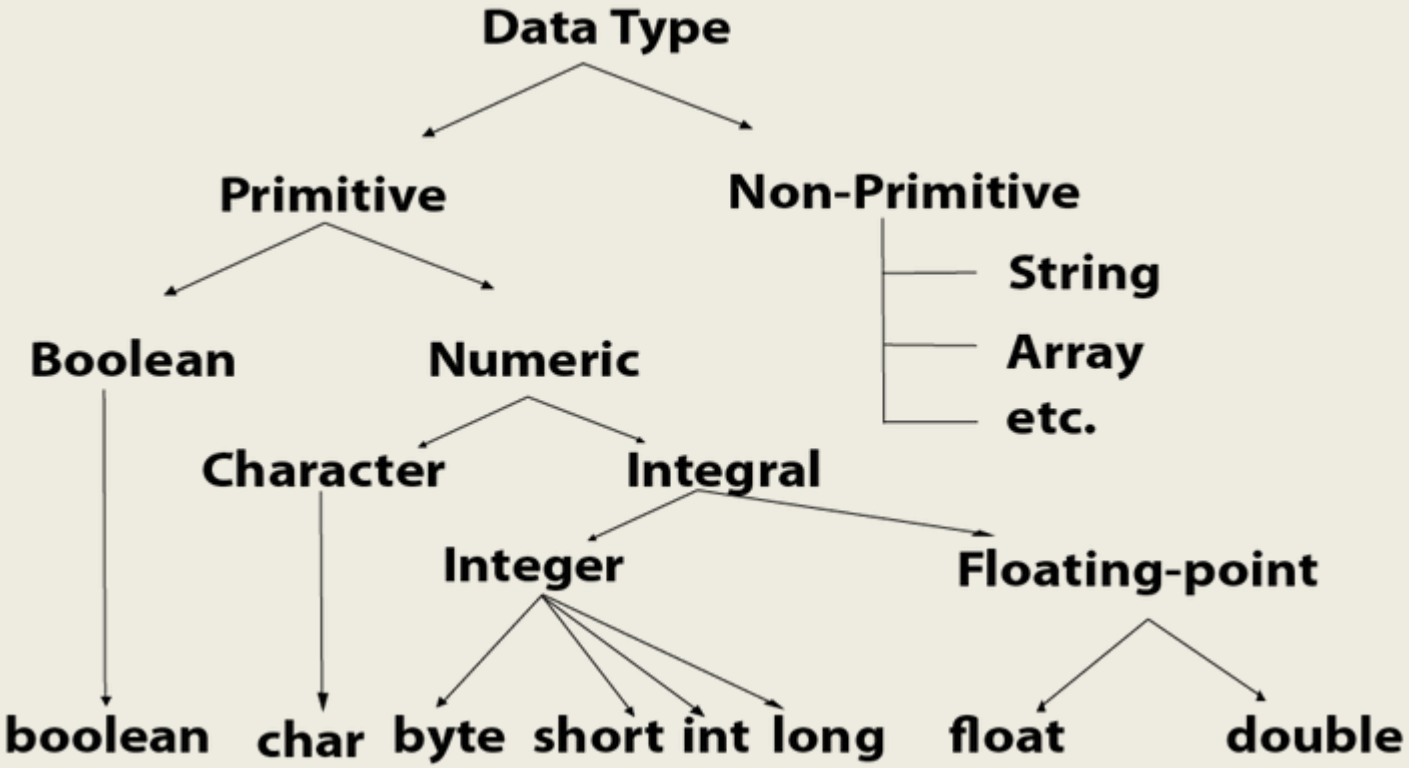
Data Types in Java

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

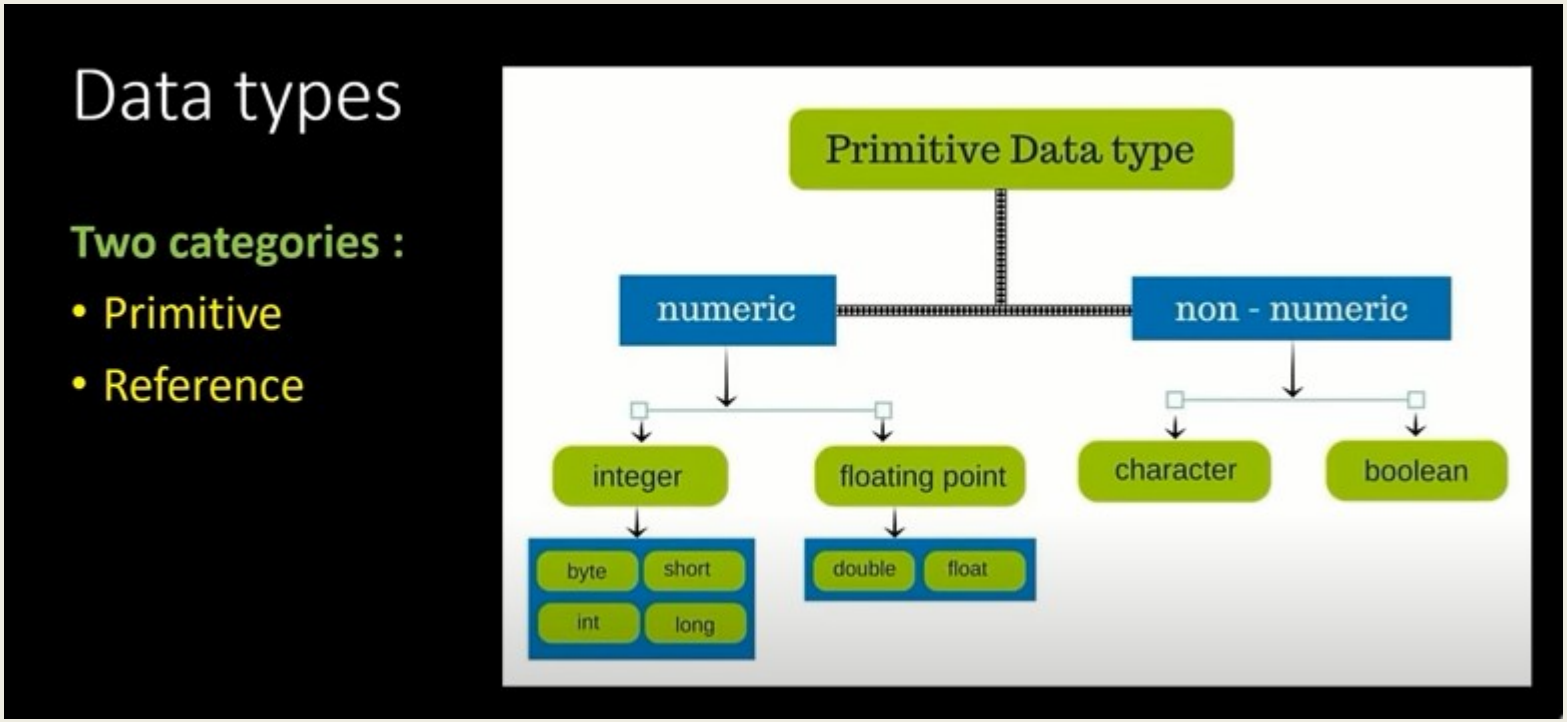
Primitive data types: The primitive data types include boolean, char, byte, short, int, long, float and double.

Non-primitive data types: The non-primitive data types include Classes

Data Types in Java



Data Types in Java



KGiSL MicroCollege

Data Types in Java

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

KGiSL MicroCollege

Data Types in Java

Non-Primitive Data Types

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

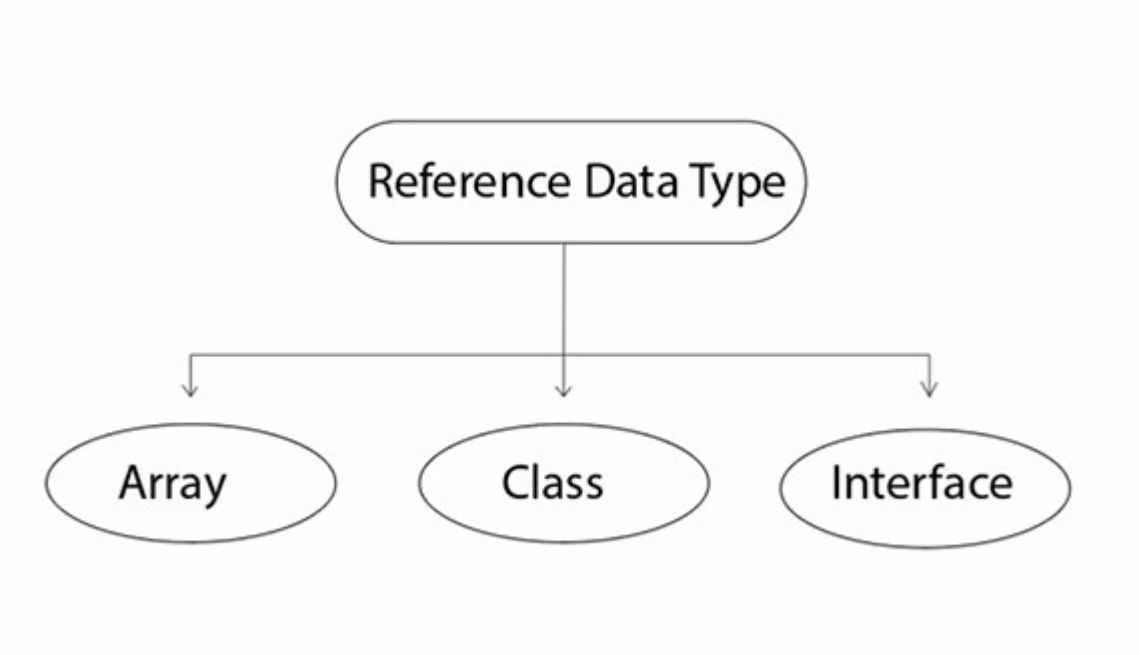
The main difference between **primitive** and **non-primitive** data types are:

- 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.
- The size of a primitive type depends on the data type, while non-primitive types have all the same size.

Examples of non-primitive types are Strings, Arrays, Classes, Interface, etc. You will learn more about these in a later chapter.

KGiSL MicroCollege

Data Types in Java



Data Types in Java

```java
public static void main(String[] args) {
// write your code here
    byte number = 20 ;
    short number2 = 150;
    int number3 = 1999;
    long number4 = 123456789789L;
    float number5 = 11.5F;
    double number6 = 1111.99999999;
    char alphabet = 'b' ;
    boolean bool = false ;

    System.out.println(bool);
```

KGiSL MicroCollege

Data Types in Java

Primitive Data Types - float

The below program declares a float variable called **price** and assigns a value of **20.12** to it.
The **f in the end** indicates it is a float data type.

```
public class Hello {

    public static void main(String[] args) {
        float price=20.12f;
        System.out.println(price);
    }
}
```

Data Types in Java

Primitive Data Types - double

The below program declares a double variable called **hike** and assigns a value of **120.12** to it.

```
public class Hello {

    public static void main(String[] args) {
        double hike=120.12;
        System.out.println(hike);
    }
}
```

By default any floating point value is considered as double but you can also specify d in the end to indicate that the data type is double.

Hence **double hike=120.12;** and **double hike=120.12d;** are same.

Data Types in Java

Accepting Input using Scanner - int data type

Till now in our examples we hardcoded values (also called literals). But in practice, the program must accept input from a source.

Scanner is an important class whose instances are useful for breaking down formatted input into tokens and translating individual tokens according to their data type.

As an example the below program reads an int data type, adds 100 to it and prints the new value as the output.

```
import java.util.Scanner;

public class Hello {

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int x = sc.nextInt();
        System.out.println(x + 100);
    }
}
```

KGiSL MicroCollege

Data Types in Java

The line **import java.util.Scanner;** denotes the package to find Scanner class. ***We will learn about packages in later chapters.***

Here the instance of the Scanner is created using the standard input (which is the console).

To accept a double we use **nextDouble()** method.

To accept a char we do not have nextChar but use the next() method and retrieve the first character as shown below.

```
char c = sc.next().charAt(0);
```

KGiSL MicroCollege

Data Types in Java

Accepting Input using Scanner - double data type

To accept real (floating) point values, we use **nextDouble()** method of Scanner as shown below.

```
import java.util.Scanner;

public class Hello {

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        double val = sc.nextDouble();
        System.out.println(val);
    }
}
```

The above program just accepts a double and prints the input value as the output.

KGiSL MicroCollege

Data Types in Java

Formatting output - decimal places

When printing data types like double and float as output, we can format the values upto certain decimal places.

As an example, the below program rounds up the output value upto 2 decimal places.

```
import java.util.Scanner;

public class Hello {

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        double price = sc.nextDouble();
        System.out.format("%.2f", price);
    }
}
```

If the input to the above program is 12.2566, the output is 12.26 (rounded upto 2 decimal places).
If the input to the above program is 5.1249, the output is 5.12 (rounded upto 2 decimal places).

KGiSL MicroCollege

Variables

- **double**: for floating-point or real numbers with optional decimal points and fractional parts in fixed or scientific notations, such as 3.1416, -55.66.
- **String**: for texts such as "Hello" or "Good Morning!". Text strings are enclosed within double quotes.

You can declare a variable of a type and assign it a value.
Example: String name = "David";

JAVA

T e called **name** of
t gns it the value "David".

Variables

Which variable type would you use for a city name?

a. Integer
b. Float
c. String

KGiSL MicroCollege

Variables

Examples of variable declarations:

```
class MyClass {
  public static void main(String[ ] args) {
    String name ="David";
    int age = 42;
    double score =15.9;
    char group = 'Z';
  }
}
```

char stands for character and holds a single character.

- Another type is the **Boolean** type, which has only two possible values: **true** and **false**.
- This data type is used for simple flags that track true/false conditions.

For example:

```
boolean online = true;
```

Variables

Variables store data for processing.

- A variable is given a name (or **identifier**), such as area, age, height, and the like.
- The name uniquely identifies each variable, assigning a value to the variable and retrieving the value stored.

- Variables have **types**. Some examples:

- **int**: for integers (whole numbers) such as 123 and -456

Java Keywords

- Java keywords are also known as reserved words.
- Keywords are particular words that act as a key to a code.
- These are predefined words by Java so they cannot be used as a variable or object name or class name.

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

Comments

The purpose of including comments in your code is to explain what the code is doing. Java supports both single and multi-line comments. All characters that appear within a comment are ignored by the Java compiler.

A **single-line** comment starts with **two forward slashes** and continues until it reaches the end of the line.

For example:

```
// this is a single-line comment
x = 5; // a single-line comment after code
```

KGiSL MicroCollege

Single-line comments are created using:

```
** characters at the beginning of the line
```

```
// characters at the end of the line
```

```
*/ characters at the beginning of the line
```

```
// characters at the beginning of the line
```

Single-line comments are created using:

```
** characters at the beginning of the line
```

```
// characters at the end of the line
```

```
*/ characters at the beginning of the line
```

```
// characters at the beginning of the line
```

Multi-Line Comments

Java also supports comments that span multiple lines.
You start this type of comment with a forward slash followed by an asterisk, and end it with an asterisk followed by a forward slash.

For example:

```
/*  This is also a
    comment spanning
    multiple lines */
```

Note that Java does not support nested multi-line comments.
However, you can nest single-line comments within multi-line comments

```
/* This is a single-line comment:

    // a single-line comment

 */
```

Multi-Line Comments

Java also supports comments that span multiple lines.
You start this type of comment with a forward slash followed by an asterisk, and end it with an asterisk followed by a forward slash.

For example:

```
/*  This is also a
    comment spanning
    multiple lines */
```

Note that Java does not support nested multi-line comments.
However, you can nest single-line comments within multi-line comments

```
/* This is a single-line comment:

    // a single-line comment

 */
```

Documentation Comments

Documentation comments are special comments that have the appearance of multi-line comments, with the difference being that they generate external documentation of your source code.

These begin with a forward slash followed by two asterisks, and end with an asterisk followed by a forward slash.

For example:

```
/** This is a documentation comment */

/** This is also a
    documentation comment */
```

Javadoc is a tool which comes with JDK and it is used for generating Java code documentation in HTML format from Java source code which has required documentation in a predefined format.

When a documentation comment begins with more than two asterisks, Javadoc assumes that you want to create a "box" around the comment in the source code. It simply ignores the extra asterisks.

For example:

```
/**********************

This is the start of a method

**********************/
```

Getting User Input

While Java provides many different methods for getting user input, the **Scanner** object is the most common, and perhaps the easiest to implement.

Import the **Scanner** class to use the **Scanner** object, as seen here:

```
import java.util.Scanner;
```

In order to use the **Scanner** class, create an instance of the class by using the following syntax:

```
Scanner myVar = new Scanner(System.in);
```

You can now read in different kinds of input data that the user enters.
Here are some methods that are available through the Scanner class:
Read a byte - nextByte()
Read a short - nextShort()
Read an int - nextInt()
Read a long - nextLong()
Read a float - nextFloat()
Read a double - nextDouble()
Read a boolean - nextBoolean()
Read a complete line - nextLine()
Read a word - next()

Example of a program used to get user input:

```
import java.util.Scanner;

class MyClass {
  public static void main(String[ ] args) {
    Scanner myVar = new Scanner(System.in);
    System.out.println(myVar.nextLine());
  }
}
```

Primitive Operators

Operators in Java

Operator in Java is a symbol that is used to perform operations. For example: +, -, *, / etc.

There are many types of operators in Java which are given below:

- **Unary Operator,**
- **Arithmetic Operator,**
- **Shift Operator,**
- **Relational Operator,**
- **Bitwise Operator,**
- **Logical Operator,**
- **Ternary Operator and**
- **Assignment Operator.**

Java Operator Precedence

Java Operator Precedence Table

Precedence	Operator	Type	Associativity
15	() [] .	Parentheses Array subscript Member selection	Left to Right
14	++ --	Unary post-increment Unary post-decrement	Right to left
13	++ -- + - ! ~ (*type*)	Unary pre-increment Unary pre-decrement Unary plus Unary minus Unary logical negation Unary bitwise complement Unary type cast	Right to left
12	* / %	Multiplication Division Modulus	Left to right
11	+ -	Addition Subtraction	Left to right
10	<< >> >>>	Bitwise left shift Bitwise right shift with sign extension Bitwise right shift with zero extension	Left to right
9	< <= > >= instanceof	Relational less than Relational less than or equal Relational greater than Relational greater than or equal Type comparison (objects only)	Left to right
8	== !=	Relational is equal to Relational is not equal to	Left to right
7	&	Bitwise AND	Left to right
6	^	Bitwise exclusive OR	Left to right
5	|	Bitwise inclusive OR	Left to right
4	&&	Logical AND	Left to right
3	||	Logical OR	Left to right
2	? :	Ternary conditional	Right to left
1	= += -= *= /= %=	Assignment Addition assignment Subtraction assignment Multiplication assignment Division assignment Modulus assignment	Right to left

Larger number means higher precedence.

KGiSL MicroCollege

Java Operator Precedence

Operator Type	Category	Precedence
Unary	**postfix**	***expr*++ *expr*--**
	prefix	**++*expr* --*expr* +*expr* -*expr* ~ !**
Arithmetic	**multiplicative**	*** / %**
	additive	**+ -**
Shift	**shift**	**<< >> >>>**
Relational	**comparison**	**< > <= >= instanceof**
	equality	**== !=**
Bitwise	**bitwise AND**	**&**
	bitwise exclusive OR	**^**
	bitwise inclusive OR	**|**
Logical	**logical AND**	**&&**
	logical OR	**||**
Ternary	**ternary**	**? :**
Assignment	**assignment**	**= += -= *= /= %= &= ^= |= <<= >>= >>>=**

Java Unary Operator

The Java unary operators require only one operand.
Unary operators are used to perform various operations i.e.:

- incrementing/decrementing a value by one
- negating an expression
- inverting the value of a boolean

```
public class OperatorExample{
public static void main(String args[]){
int x=10;
System.out.println(x++);//10 (11)
System.out.println(++x);//12
System.out.println(x--);//12 (11)
System.out.println(--x);//10
}}
```

```
public class OperatorExample{
public static void main(String args[]){
int a=10;
int b=10;
System.out.println(a++ + ++a);//10+12=22
System.out.println(b++ + b++);//10+11=21

}}
```

```
public class OperatorExample{
public static void main(String args[]){
int a=10;
int b=-10;
boolean c=true;
boolean d=false;
System.out.println(~a);//-11 (minus of total positive value which starts from 0)
System.out.println(~b);//9 (positive of total minus, positive starts from 0)
System.out.println(!c);//false (opposite of boolean value)
System.out.println(!d);//true
}}
```

KGiSL MicroCollege

Java Arithmetic Operators

Java arithmetic operators are used to perform addition, subtraction, multiplication, and division. They act as basic mathematical operations.

```
public class OperatorExample{
public static void main(String args[]){
int a=10;
int b=5;
System.out.println(a+b);//15
System.out.println(a-b);//5
System.out.println(a*b);//50
System.out.println(a/b);//2
System.out.println(a%b);//0
}}
```

```
public class OperatorExample{
public static void main(String args[]){
System.out.println(10*10/5+3-1*4/2);
}}
```

Java Arithmetic Operators

Arithmetic Operators

+ **Additive operator (also used for String concatenation)**
- **Subtraction operator**
* **Multiplication operator**
/ **Division operator**
% **Remainder operator**

The below program prints the product of two int values x and y (passed as input) using the multiplication operator *.

```
import java.util.Scanner;

public class Hello {

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int x=sc.nextInt();
        int y=sc.nextInt();
        System.out.println(x*y);
    }
}
```

KGiSL MicroCollege

Java Left Shift Operator

The Java left shift operator << is used to shift all of the bits in a value to the left side of a specified number of times.

```
public class OperatorExample{

public static void main(String args[]){

System.out.println(10<<2);//10*2^2=10*4=40

System.out.println(10<<3);//10*2^3=10*8=80

System.out.println(20<<2);//20*2^2=20*4=80

System.out.println(15<<4);//15*2^4=15*16=240

}}
```

Java Right Shift Operator

The Java right shift operator >> is used to move the value of the left operand to right by the number of bits specified by the right operand.

```
public OperatorExample{

public static void main(String args[]){

System.out.println(10>>2);//10/2^2=10/4=2

System.out.println(20>>2);//20/2^2=20/4=5

System.out.println(20>>3);//20/2^3=20/8=2

}}
```

Java Shift Operator Example: >> vs >>>

```
public class OperatorExample{
public static void main(String args[]){
    //For positive number, >> and >>> works same
    System.out.println(20>>2);
    System.out.println(20>>>2);
    //For negative number, >>> changes parity bit (MSB) to 0
    System.out.println(-20>>2);
    System.out.println(-20>>>2);
}}
```

Java AND Operator Example: Logical && and Bitwise &

The logical && operator doesn't check the second condition if the first condition is false. It checks the second condition only if the first one is true.

The bitwise & operator always checks both conditions whether first condition is true or false.

```
public class OperatorExample{

public static void main(String args[]){

int a=10;

int b=5;

int c=20;

System.out.println(a<b&&a<c);//false && true = false

System.out.println(a<b&a<c);//false & true = false

}}
```

Java AND Operator Example: Logical && vs Bitwise &

```
public class OperatorExample{
public static void main(String args[]){
int a=10;
int b=5;
int c=20;
System.out.println(a<b&&a++<c);//false && true = false
System.out.println(a);//10 because second condition is not checked
System.out.println(a<b&a++<c);//false && true = false
System.out.println(a);//11 because second condition is checked
}}
```

Java OR Operator Example: Logical || and Bitwise |

The logical || operator doesn't check the second condition if the first condition is true. It checks the second condition only if the first one is false.

The bitwise | operator always checks both conditions whether

```
public class OperatorExample{
public static void main(String args[]){
int a=10;
int b=5;
int c=20;
System.out.println(a>b||a<c);//true || true = true
System.out.println(a>b|a<c);//true | true = true
//|| vs |
System.out.println(a>b||a++<c);//true || true = true
System.out.println(a);//10 because second condition is not checked
System.out.println(a>b|a++<c);//true | true = true
System.out.println(a);//11 because second condition is checked
}}
```

Java Ternary Operator

Java Ternary operator is used as one line replacement for if-then-else statement and used a lot in Java programming. It is the only conditional operator which takes three operands.

```
public class OperatorExample{

public static void main(String args[]){

int a=2;

int b=5;

int min=(a<b)?a:b;

System.out.println(min);

}}
```

Java Assignment Operator

Java assignment operator is one of the most common operators. It is used to assign the value on its right to the operand on its left.

```
public class OperatorExample{
public static void main(String args[]){
int a=10;
int b=20;
a+=4;//a=a+4 (a=10+4)
b-=4;//b=b-4 (b=20-4)
System.out.println(a);
System.out.println(b);
}}
```

Java Assignment Operator Example

```
public class OperatorExample{
public static void main(String[] args){
int a=10;
a+=3;//10+3
System.out.println(a);
a-=4;//13-4
System.out.println(a);
a*=2;//9*2
System.out.println(a);
a/=2;//18/2
System.out.println(a);
}}
```

KGiSL MicroCollege

The Math Operators

Java provides a rich set of operators to use in manipulating variables. A value used on either side of an operator is called an **operand**. For example, in the expression below, the numbers 6 and 3 are operands of the plus

```
int x = 6 + 3;
```

Java arithmetic operators:

+ **addition**
- **subtraction**
* **multiplication**
/ **division**
% **modulo**

KGiSL MicroCollege

Addition

The + operator adds together two values, such as two constants, a constant and a variable, or a variable and a variable. Here are a few examples of addition:

```
int sum1 = 50 + 10;
int sum2 = sum1 + 66;
int sum3 = sum2 + sum2;
```

Subtraction

The - operator subtracts one value from another.

```
int sum1 = 1000 - 10;
int sum2 = sum1 - 5;
int sum3 = sum1 - sum2;
```

KGiSL MicroCollege

Addition

The + operator adds together two values, such as two constants, a constant and a variable, or a variable and a variable. Here are a few examples of addition:

```
int sum1 = 50 + 10;
int sum2 = sum1 + 66;
int sum3 = sum2 + sum2;
```

Subtraction

The - operator subtracts one value from another.

```
int sum1 = 1000 - 10;
int sum2 = sum1 - 5;
int sum3 = sum1 - sum2;
```

KGiSL MicroCollege

Multiplication

The * operator multiplies two values.

```
int sum1 = 1000 * 2;
int sum2 = sum1 * 10;
int sum3 = sum1 * sum2;
```

Division

The / operator divides one value by another.

```
int sum1 = 1000 / 5;
int sum2 = sum1 / 2;
int sum3 = sum1 / sum2;
```

Multiplication

The * operator multiplies two values.

```
int sum1 = 1000 * 2;
int sum2 = sum1 * 10;
int sum3 = sum1 * sum2;
```

Division

The / operator divides one value by another.

```
int sum1 = 1000 / 5;
int sum2 = sum1 / 2;
int sum3 = sum1 / sum2;
```

Modulo

The **modulo** (or remainder) math operation performs an integer division of one value by another, and returns the remainder of that division.
The operator for the modulo operation is the percentage (%) character.

Examples

```
int value = 23;
int res = value % 6; // res is 5
```

Increment Operators

An **increment** or **decrement** operator provides a more convenient and compact way to increase or decrease the value of a variable by **one**.

For example, the statement **x=x+1;** can be simplified to **++x;**

Example:

```
int test = 5;
++test; // test is now 6
```

The decrement operator (--) is used to decrease the value of a variable by one.

```
int test = 5;
--test; // test is now 4
```

KGiSL MicroCollege

Increment Operators

An **increment** or **decrement** operator provides a more convenient and compact way to increase or decrease the value of a variable by **one**.

For example, the statement **x=x+1;** can be simplified to **++x;**

Example:

```
int test = 5;
++test; // test is now 6
```

The decrement operator (--) is used to decrease the value of a variable by one.

```
int test = 5;
--test; // test is now 4
```

KGiSL MicroCollege

Prefix & Postfix

- Two forms, **prefix** and **postfix**, may be used with both the increment and decrement operators.
- With prefix form, the operator appears before the operand, while in postfix form, the operator appears after the operand.
- Below is an explanation of how the two forms work:
  Prefix: Increments the variable's value and uses the new value in the expression.

```
int x = 34;
int y = ++x; // y is 35
```

The value of x is first incremented to 35, and is then assigned to y, so the values of both x and y are now 35.

Postfix: The variable's value is first used in the expression and is then increased.

Example:

```
int x = 34;
int y = x++; // y is 34
```

x is first assigned to y, and is then incremented by one. Therefore, x becomes 35, while y is assigned the value of 34.

Assignment Operators

You are already familiar with the assignment operator (=), which assigns a value to a variable.

```
int value = 5;
```

This assigned the value 5 to a variable called **value** of type **int**.

Java provides a number of assignment operators to make it easier to write code.

```
int num1 = 4;
int num2 = 8;
num2 += num1; // num2 = num2 + num1;

// num2 is 12 and num1 is 4
```

Subtraction and assignment (-=):

```
int num1 = 4;
int num2 = 8;
num2 -= num1; // num2 = num2 - num1;

// num2 is 4 and num1 is 4
```

Java Control Statements

Java Control Statements | Control Flow in Java

Java compiler executes the code from top to bottom.

The statements in the code are executed according to the order in which they appear.

However, **Java provides statements that can be used to control the flow of Java code.** Such statements are called control flow statements.

It is one of the fundamental features of Java, which provides a smooth flow of program.

Java provides three types of control flow statements.

Decision Making statements

- **if statements**
- **switch statement**

Loop statements

- **do while loop**
- **while loop**
- **for loop**
- **for-each loop**

Jump statements

- **break statement**
- **continue statement**

Java if Statement

- The Java *if statement* is used to test the condition.
- It checks boolean condition: *true* or *false*.

There are various types of if statement in Java.

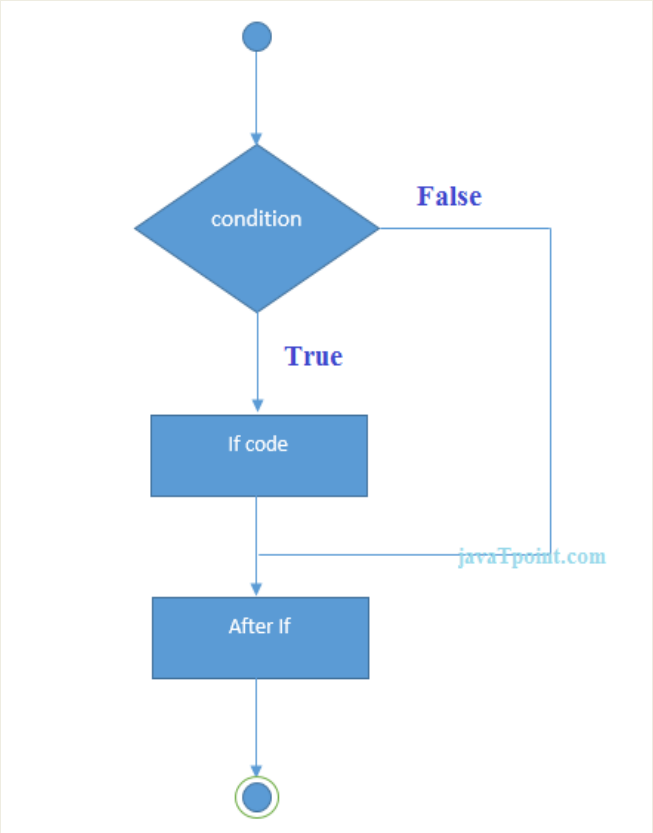
1. if statement
2. if-else statement
3. if-else-if ladder
4. nested if statement

Java if Statement

The Java if statement tests the condition. **It executes the *if block* if condition is true.**

Syntax:

```
if(condition){
//code to be executed
}
```




```
public class IfExample {
public static void main(String[] args) {
   //defining an 'age' variable
   int age=20;
   //checking the age
   if(age>18){
      System.out.print("Age is greater than 18");
   }
}
}
```

KGiSL MicroCollege

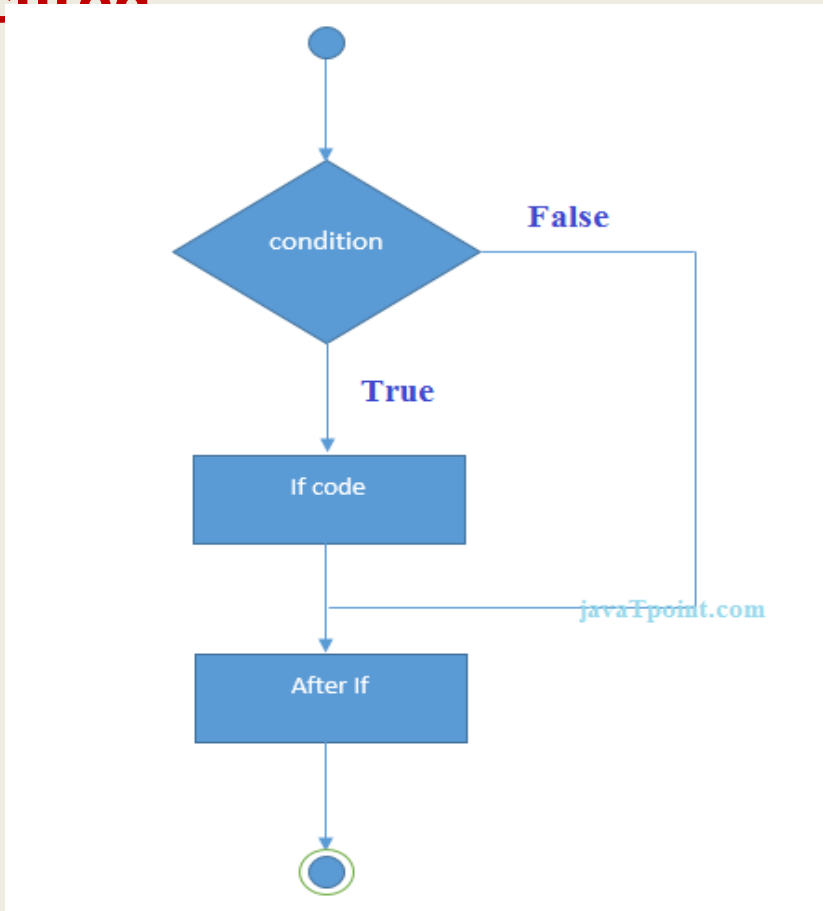
Java if-else Statement

The Java if-else statement also tests the condition. **It executes the *if block* if condition is true otherwise *else block* is executed**

Syntax:

```
if(condition){
//code if condition is true
}else{
```

```
//It is a program of odd and even number.
public class IfElseExample {
public static void main(String[] args) {
   //defining a variable
   int number=13;
   //Check if the number is divisible by 2 or not
   if(number%2==0){
      System.out.println("even number");
   }else{
      System.out.println("odd number");
   }
}
}
```



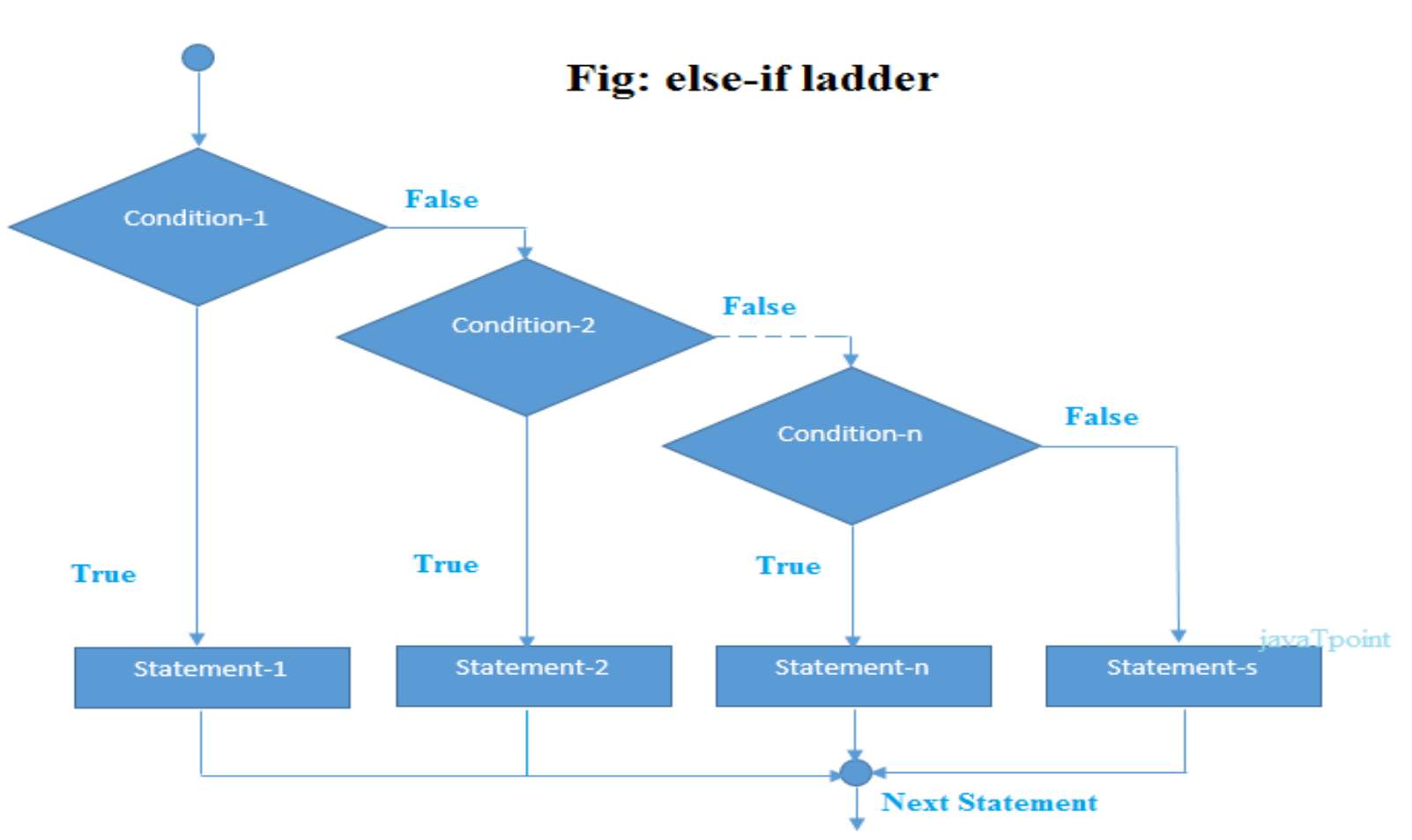
Java if-else-if ladder Statement

The if-else-if ladder statement executes one condition from multiple statements.

Syntax:

```
if(condition1){
//code to be executed if condition1 is true
}else if(condition2){
//code to be executed if condition2 is true
}
else if(condition3){
//code to be executed if condition3 is true
}
...
else{
//code to be executed if all the conditions are false
}
```

Java if-else-if ladder Statement



Java if-else-if ladder Statement

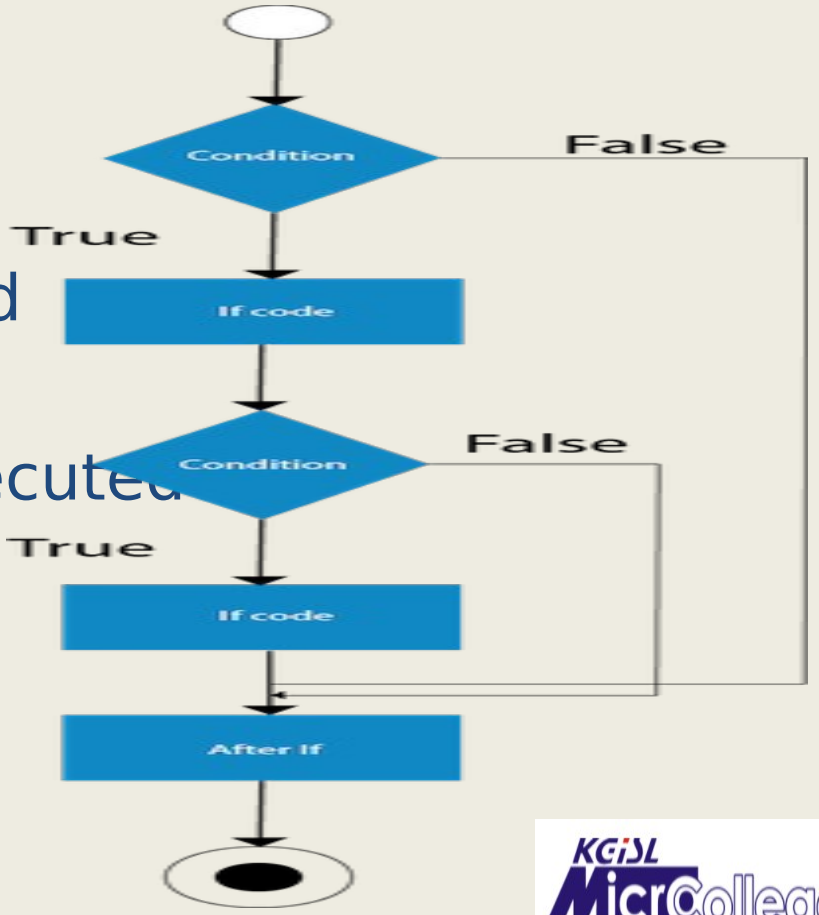
```
public class IfElseIfExample {
public static void main(String[] args) {
    int marks=65;
    if(marks<50){
        System.out.println("fail");
    }
    else if(marks>=50 && marks<60){
        System.out.println("D grade");
    }
    else if(marks>=60 && marks<70){
        System.out.println("C grade");
    }
    else if(marks>=70 && marks<80){
        System.out.println("B grade");
    }
    else if(marks>=80 && marks<90){
        System.out.println("A grade");
    }else if(marks>=90 && marks<100){
        System.out.println("A+ grade");
    }else{
        System.out.println("Invalid!");
    }
}
}
```

Java Nested if statement

The nested if statement represents the *if block within another if block*. Here, the inner if block condition executes only when outer if block condition is true.

Syntax:

```
if(condition){
    //code to be executed
        if(condition){
            //code to be executed
    }
}
```



Java Nested if statement

```
//
Java Program to demonstrate the use of Nested If State
ment.
public class JavaNestedIfExample {
public static void main(String[] args) {
    //Creating two variables for age and weight
    int age=20;
    int weight=80;
    //applying condition on age and weight
    if(age>=18){
        if(weight>50){
            System.out.println("You are eligible to donate blo
od");
        }
    }
}}
```

Java Switch Statement

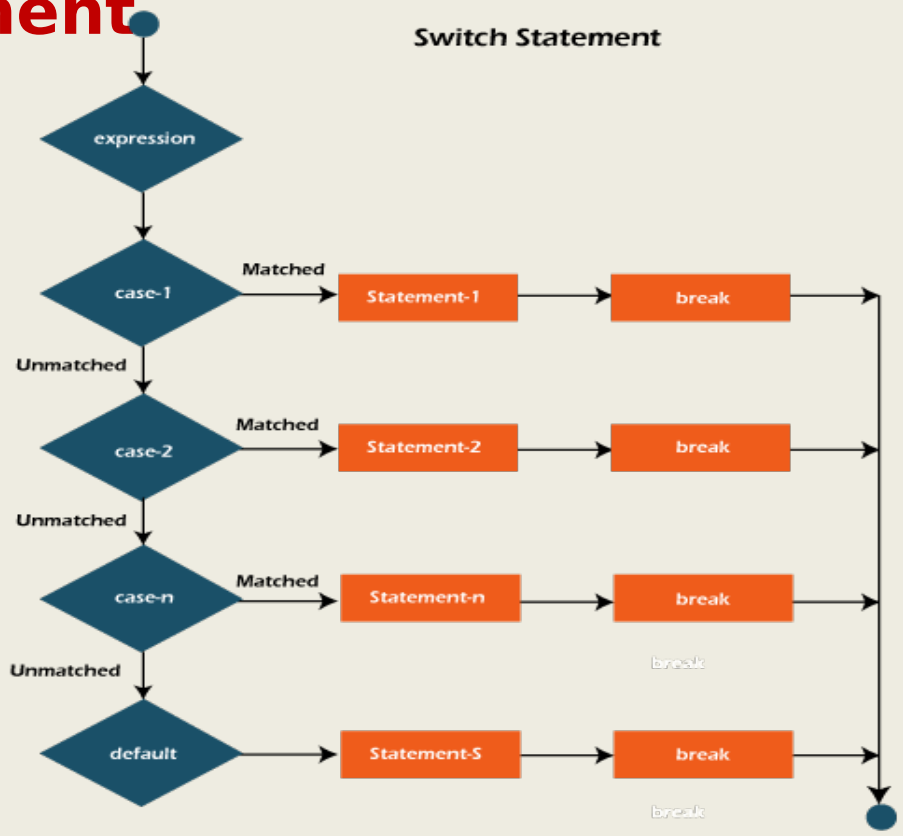
- The Java *switch statement* executes one statement from multiple conditions. It is like if-else-if ladder statement.
- There can be ***one or N number of case*** *values* for a switch expression.
- The case value must be of switch expression type only. The case value must be *literal or constant*. It doesn't allow variables.
- The case values must be *unique*. In case of duplicate value, it renders compile-time error.
- The Java switch expression must be of *byte, short, int, long (with its Wrapper type), enums and string*.
- Each case statement can have a *break statement* which is optional. When control reaches to the break statement, it jumps the control after the switch expression. If a break statement is not found, it executes the next case.
- The case value can have a *default label* which is optional.

Java Switch Statement

Syntax:

```
switch(expression){
case value1:
 //code to be executed;
 break;  //optional
case value2:
 //code to be executed;
 break;  //optional
.....

default:
  code to be executed if all cases are not matched;
}
```



Java Switch Statement

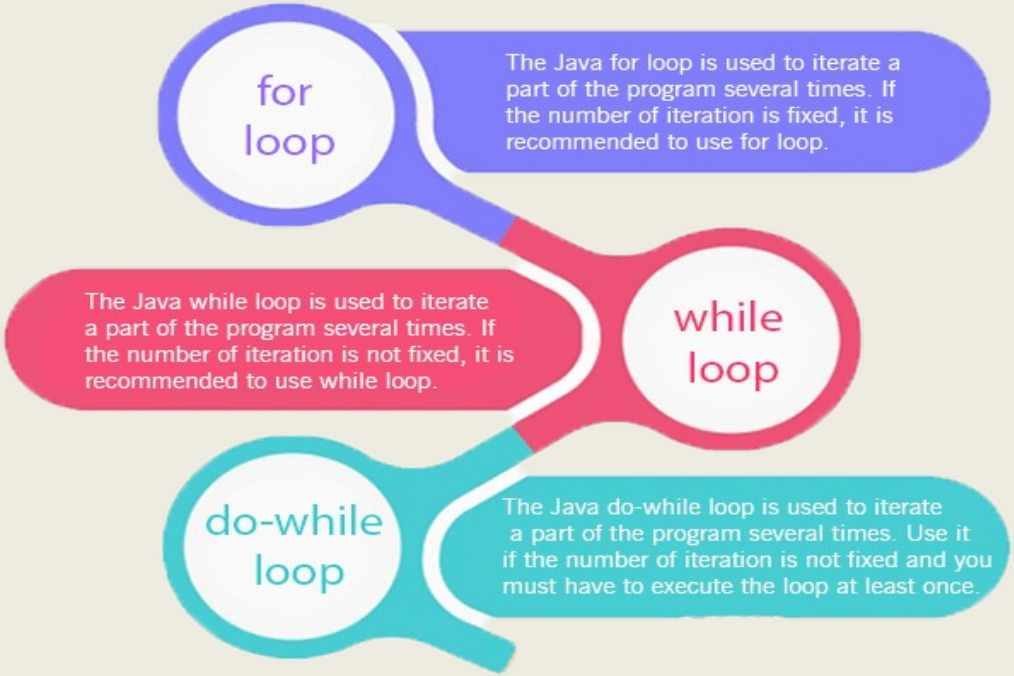
```
public class SwitchExample {
public static void main(String[] args) {
    //Declaring a variable for switch expression
    int number=20;
    //Switch expression
    switch(number){
    //Case statements
    case 10: System.out.println("10");
    break;
    case 20: System.out.println("20");
    break;
    case 30: System.out.println("30");
    break;
    //Default case statement
    default:System.out.println("Not in 10, 20 or 30");
    }
}
}
```

Loops in Java

Java Loops:

- The Java *for loop* is used to iterate a part of the program several times.
- If the number of iteration is **fixed**, it is recommended to use for loop.

There are three types of for loops in Java.



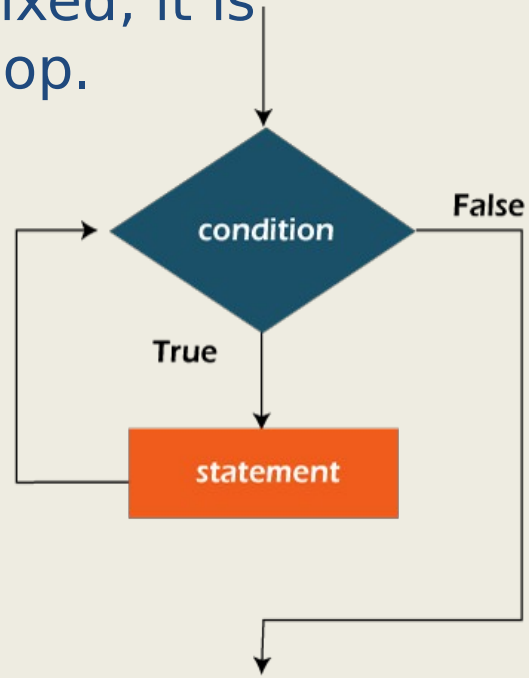
KGiSL MicroCollege

Java While Loop

- The Java *while loop* is used to iterate a part of the program repeatedly until the specified Boolean condition is true.
- As soon as the Boolean condition becomes false, the loop automatically stops.
- The while loop is considered as a repeating if statement.
- If the number of iteration is not fixed, it is recommended to use the while loop.

Syntax:

```
while (condition){
//code to be executed
Increment / decrement statement
}
```



KGiSL MicrCollege

Java While Loop

```
public class WhileExample {
public static void main(String[] args
) {
    int i=1;
    while(i<=10){
        System.out.println(i);
    i++;
    }
}
}
```

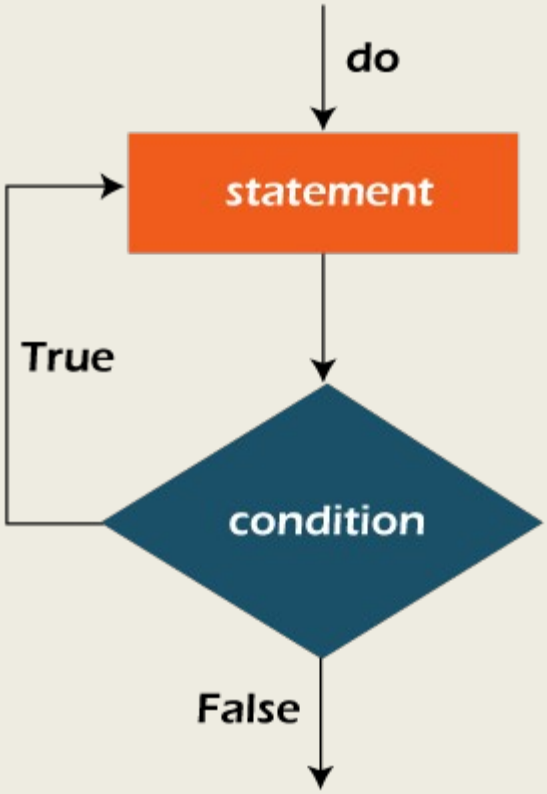
do-while loop

- The Java *do-while loop* is used to iterate a part of the program repeatedly, until the specified condition is true.
- If the number of iteration is not fixed and you must have to execute the loop at least once, it is recommended to use a do-while loop.
- Java do-while loop is called an **exit control loop**. Therefore, unlike while loop and for loop, the do-while check the condition at the end of loop body.
- The Java *do-while loop* is executed at least once because condition is checked after loop body.

Syntax:

```
do{
//code to be executed / loop body
//update statement
}while (condition);
```

do-while loop




```
public class DoWhileExample
{
public static void main(Strin
g[] args)
 {
    int i=1;
    do{
       System.out.println(i);
    i++;
    }while(i<=10);
}
}
```

Java Simple for Loop

A simple for loop is the same as C/C++. We can initialize the variable, check condition and increment/decrement value. It consists of four parts:

Initialization: It is the initial condition which is executed once when the loop starts. Here, we can initialize the variable, or we can use an already initialized variable. It is an optional condition.

Condition: It is the second condition which is executed each time to test the condition of the loop. It continues execution until the condition is false. It must return boolean value either true or false. It is an optional condition.

Java Simple for Loop

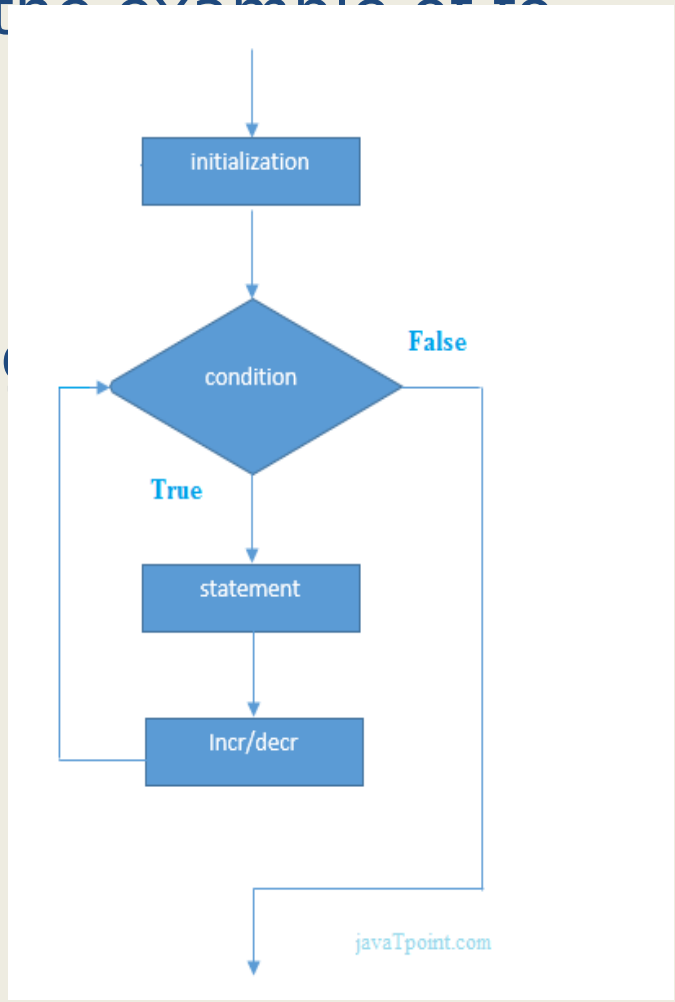
Increment/Decrement: It increments or decrements the variable value. It is an optional condition.

Statement: The statement of the loop is executed each time until the second condition is false.

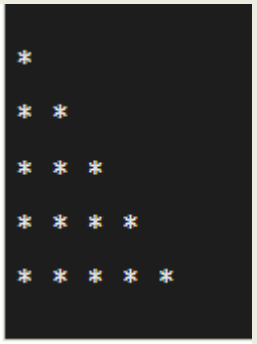
Syntax:

```
for(initialization; condition; increment/
decrement){
//statement or code to be executed    }
```

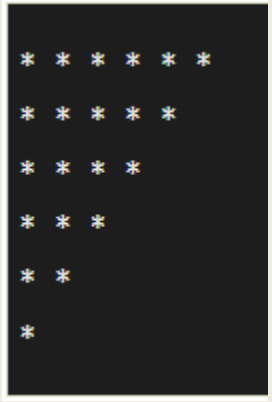
```
//
Java Program to demonstrate the example of fo
r loop
//which prints table of 1
public class ForExample {
public static void main(Strin
 {
    //Code of Java for loop
    for(int i=1;i<=10;i++){
        System.out.println(i);
    }
}
}
```



```
public class PyramidExample {
public static void main(String[] args
) {
for(int i=1;i<=5;i++){
for(int j=1;j<=i;j++){
        System.out.print("* ");
}
System.out.println();//new line
}
}
}
```




```
public class PyramidExample2 {
public static void main(String[] args
) {
int term=6;
for(int i=1;i<=term;i++){
for(int j=term;j>=i;j--){
        System.out.print("* ");
}
System.out.println();//new line
}
}
}
```



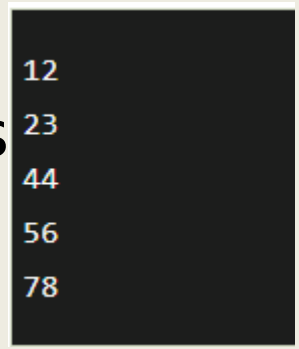
for-each Loop

The for-each loop is used to traverse array or collection in Java. It is easier to use than simple for loop because we don't need to increment value and use subscript notation. It works on the basis of elements and not the index. It returns element one by one in the defined variable.

Syntax:

```
for(data_type variable : array_name){
//code to be executed
}
```

```
//Java For-each loop example which prints the
//elements of the array
public class ForEachExample
 {
public static void main(String[] args)
{
    //Declaring an array
    int arr[]={12,23,44,56,78};
    //Printing array using for-each loop
    for(int i:arr){
        System.out.println(i);
    }
}
}
```



Java Break Statement

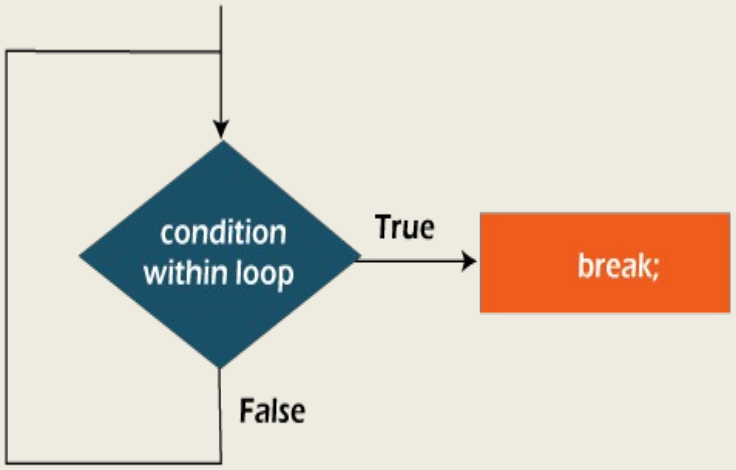
- When a break statement is encountered inside a loop, the loop is immediately terminated and the program control resumes at the next statement following the loop.
- The Java *break* statement is used to break loop or switch statement. It breaks the current flow of the program at specified condition. In case of inner loop, it breaks only inner loop.
- We can use Java break statement in all types of loops such as for loop, while loop and do-while loop.

Syntax:

jump-statement;

break;

Java Break Statement



Flowchart of break statement

```
public class BreakExample {
public static void main(String[] args
) {
    //using for loop
    for(int i=1;i<=10;i++){
        if(i==5){
            //breaking the loop
            break;
        }
        System.out.println(i);
    }
}
}
```

Continue Statement

- The continue statement is used in loop control structure when you need to jump to the next iteration of the loop immediately. It can be used with for loop or while loop.
- The Java *continue statement* is used to continue the loop. It continues the current flow of the program and skips the remaining code at the specified condition. In case of an inner loop, it continues the inner loop only.
- We can use Java continue statement in all types of loops such as for loop, while loop and do-while loop.

Syntax:

jump-statement;
continue;

```
public class ContinueExample {
public static void main(String[] args
) {
    //for loop
    for(int i=1;i<=10;i++){
        if(i==5){
            //using continue statement
            continue;//
it will skip the rest statement
        }
        System.out.println(i);
    }
}
}
```

Output:

```
1
2
3
4
6
7
8
9
10
```

Strings

A **String** is an object that represents a sequence of characters.
For example, "Hello" is a string of 5 characters.

For example:

```
String s = "SoloLearn";
```

You are allowed to define an empty string. For example, String str = "";

String Concatenation

The + (plus) operator between strings adds them together to make a new string. This process is called **concatenation**.
The resulted string is the first string put together with the second string.
For example:

```
String firstName, lastName;
firstName = "David";
lastName = "Williams";

System.out.println("My name is " + firstName +"
"+lastName);
```

The **char** data type represents a single character.

KGiSL MicroCollege

Decision Making

Conditional statements are used to perform different actions based on different conditions. The **if statement** is one of the most frequently used conditional statements.
If the **if** statement's condition expression evaluates to true, the block of code inside the **if** statement is executed. If the expression is found to be false, the first set of code after the end of the **if** statement (after the closing curly brace) is executed.

Syntax:

```
if (condition) {
   //Executes when the condition is true
}
```

KGiSL MicroCollege

String Concatenation

The + (plus) operator between strings adds them together to make a new string. This process is called **concatenation**.
The resulted string is the first string put together with the second string.
For example:

```
String firstName, lastName;
firstName = "David";
lastName = "Williams";

System.out.println("My name is " + firstName +"
"+lastName);
```

The **char** data type represents a single character.

KGiSL MicroCollege

Any of the following comparison operators may be used to form the condition:

< less than
> greater than
!= not equal to
== equal to
<= less than or equal to
>= greater than or equal to

For example:

```
int x = 7;
if(x < 42) {
   System.out.println("Hi");
}
```

Remember that you need to use two equal signs (==) to test for equality, since a single equal sign is the assignment operator.

KGiSL MicroCollege

if...else Statements

An **if** statement can be followed by an optional **else** statement, which executes when the condition evaluates to false.
For example:

```
int age = 30;

if (age < 16) {
   System.out.println("Too Young");
} else {
   System.out.println("Welcome!");
}
```

As age equals 30, the condition in the **if** statement evaluates to false and the **else** statement is executed.

KGiSL MicroCollege

Nested if Statements

You can use one **if-else** statement inside another **if** or **else** statement.

For example:

```
int age = 25;
if(age > 0) {
   if(age > 16) {
      System.out.println("Welcome!");
   } else {
      System.out.println("Too Young");
   }
} else {
   System.out.println("Error");
}
```

You can nest as many **if-else** statements as you want.

KGiSL MicroCollege

else if Statements

Instead of using nested **if-else** statements, you can use the **else if** statement to check multiple conditions.

For example:

```
int age = 25;

if(age <= 0) {
   System.out.println("Error");
} else if(age <= 16) {
   System.out.println("Too Young");
} else if(age < 100) {
   System.out.println("Welcome!");
} else {
   System.out.println("Really?");
}
```

The code will check the condition to evaluate to true and execute the statements inside that block.

You can include as many **else if** statements as you need.

KGiSL MicroCollege

Logical Operators

Logical operators are used to combine multiple conditions.
Let's say you wanted your program to output "Welcome!" only when the variable **age** is greater than 18 and the variable **money** is greater than 500.

```
if (age > 18) {
   if (money > 500) {
      System.out.println("Welcome!");
   }
}
```

s is to use

Logical Operators

However, using the **AND** logical operator (**&&**) is a better way:

```
if (age > 18 && money > 500) {
   System.out.println("Welcome!");
}
```

If both operands of the AND operator are true, then the condition becomes true.

Logical Operators

The OR Operator

The **OR** operator (||) checks if any one of the conditions is true.
The condition becomes true, if any one of the operands evaluates to true.
For example:

```
int age = 25;
int money = 100;

if (age > 18 || money > 500) {
   System.out.println("Welcome!");
}
```

KGiSL MicroCollege

Logical Operators

The OR Operator

The code above will print "Welcome!" if age is greater than 18 **or** if money is greater than 500.

The **NOT !**) logical operator is used to reverse the logical state of its operand. If a condition is true, the **NOT** logical operator will make it f

```
int age = 25;
if(!(age > 18)) {
   System.out.println("Too Young");
} else {
   System.out.println("Welcome");
}
```

KGiSL MicroCollege

!(age > 18) reads as "if age is NOT greater than 18".

The switch Statement

A **switch** statement tests a variable for equality against a list of values. Each value is called a **case**, and the variable being switched on is checked for each case.

Syntax:

```
switch (expression) {
   case value1 :
     //Statements
     break; //optional
   case value2 :
     //Statements
     break; //optional
    //You can have any number of case statements.
    default : //Optional
       //Statements
}
```

The switch Statement

- When the variable being switched on is equal to a **case**, the statements following that **case** will execute until a **break** statement is reached.
- When a **break** statement is reached, the **switch** terminates, and the flow of control jumps to the next line after the **switch** statement.
- Not every **case** needs to contain a **break**. If no **break** appears, the flow of control will fall through to subsequent cases until a **break** is reached.

T ests **day** against a set of v rresponding message.

The switch Statement

```
int day = 3;

switch(day) {
  case 1:
    System.out.println("Monday");
    break;
  case 2:
    System.out.println("Tuesday");
    break;
  case 3:
    System.out.println("Wednesday");
    break;
}
```

You can have any number of **case** statements within a **switch**. Each **case** is followed by the comparison value and a colon.

The default Statement

A switch statement can have an optional **default** case.

The **default** case can be used for performing a task when none of the cases is matched.

For example:

```
int day = 3;

switch(day) {
  case 6:
    System.out.println("Saturday");
    break;
  case 7:
    System.out.println("Sunday");
    break;
  default:
    System.out.println("Weekday");
}
```

No **break** is needed in the default case, as it is always the last statement in the switch.

KGiSL MicroCollege

The switch Expression

The **switch expression** allows multiple comma-separated values per case and returns a value for the whole switch-case block.

For example:

```
String dayType  = switch(day) {
            case 1, 2, 3, 4, 5 -> "Working day";
            case 6, 7 -> "Weekend";
            default -> "Invalid day";
            };
```

The switch expression makes the switch-case block much shorter and doesn't use a break statement.

Notice the **-> shorthand** after the cases.

while Loops

A **loop** statement allows to repeatedly execute a statement or group of statements.

A **while** loop statement repeatedly executes a target statement as long as a given condition is true.

```
int x = 3;

while(x > 0) {
   System.out.println(x);
   x--;
}
```

The **while** loops check for the condition x > 0. If it evaluates to true, it executes the statements within its body. Then it checks for the statement again and repeats.

while Loops

When the expression is tested and the result is false, the loop body is skipped and the first statement after the while loop is executed.

Example:

```
while( x < 10 )
{
  System.out.println(x);
  x++;
}
System.out.println("Loop ended");

/*
6
7
8
9
Loop ended
*/
```

Notice that the last print method is out of the while scope.

KGiSL MicroCollege

for Loops

Another loop structure is the **for** loop. A for loop allows you to efficiently write a loop that needs to execute a specific number of times.

Syntax:

```
for (initialization; condition; increment/decrement) {
   statement(s)
}
```

Initialization: Expression executes only once during the beginning of loop
Condition: Is evaluated each time the loop iterates. The loop executes the statement repeatedly, until this condition returns false.
Increment/Decrement: Executes after each iteration of the loop.

The following example prints the numbers 1 through 5.

```
for(int x = 1; x <=5; x++) {
  System.out.println(x);
}
```

This initializes x to the value 1, and repeatedly prints the value of x, until the condition x<=5 becomes false.

On each iteration, the statement x++ is executed, incrementing x by one.

for Loops

You can have any type of condition and any type of increment statements in the for loop.
The example below prints only the even values between 0 and 10:

```
for(int x=0; x<=10; x=x+2) {
  System.out.println(x);
}
/*
0
2
4
6
8
10
*/
```

A **for** loop is best when the starting and ending numbers are known.

do...while Loops

A **do...while** loop is similar to a **while** loop, except that a **do...while** loop is guaranteed to execute at least one time.
Example:

```
int x = 1;
do {
  System.out.println(x);
  x++;
} while(x < 5);

/*
1
2
3
4
*/
```

do...while Loops

Notice that the condition appears at the end of the loop, so the statements in the loop execute once before it is tested.
Even with a false condition, the code will run once.
Example:

```
int x = 1;
do {
  System.out.println(x);
  x++;
} while(x < 0);
```

Notice that in do...while loops, the while is just the condition and doesn't have a body itself.

Loop Control Statements

The **break** and **continue** statements change the loop's execution flow.
The **break** statement terminates the loop and transfers execution to the statement immediately following the loop.
Example:

```
int x = 1;

while(x > 0) {
 System.out.println(x);
  if(x == 4) {
    break;
  }
  x++;
}
```

Loop Control Statements

The **continue** statement causes the loop to skip the remainder of its body and then immediately retest its condition prior to reiterating. In other words, it makes the loop skip to its next iteration.

Example:

```
for(int x=10; x<=40; x=x+10) {
  if(x == 30) {
    continue;
  }
  System.out.println(x);
}
```

As you can see, the above code skips the value of 30, as directed by the **continue** statement.

KGiSL MicroCollege

Loop Control Statements

The **continue** statement causes the loop to skip the remainder of its body and then immediately retest its condition prior to reiterating. In other words, it makes the loop skip to its next iteration.

Example:

```
for(int x=10; x<=40; x=x+10) {
  if(x == 30) {
    continue;
  }
  System.out.println(x);
}
```

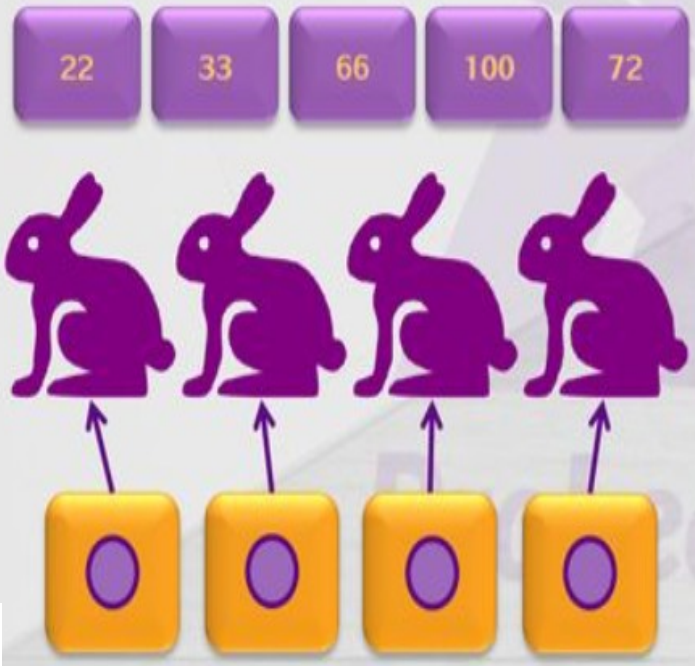
As you can see, the above code skips the value of 30, as directed by the **continue** statement.

KGiSL MicroCollege

ARRAYS

Arrays in Java

- An array is collection of elements of similar type
- Array elements are always stored in consecutive memory blocks
- Arrays could be of primitive data types or reference type
- Arrays in Java are also objects



Arrays in Java

- Reference variables are used in Java to store the references of objects created by the operator – new
- Any one of the following syntax can be used to create a reference to an int array

```
int x[ ];
int [ ] x;
```

- Th n be us rring to any int array


```
// Declart a reference to an int array
int [ ] x;
// Create a new int array and make x refer to it
x = new int [5];
```

KGiSL MicroCollege

Arrays in Java

- The following statement also creates a new int array and assigns its reference to x

```
int [ ] x = new int [5];
```

- In simple terms, references can be seen as names of an array

An array is a groups like typed variables that is referred to a by a common type name. A specific element in an array is accessed by its index. Array offers a convenient meaning of grouping same information.

Arrays in Java

The entire array has a single name

Each value has a numeric index

An array of size N is indexed from zero to N-1

This array holds 10 values that are indexed from 0 to 9

Arrays in Java

- A particular value in an array is referenced

  using the array name followed by the index in brackets

- For example, the expression

  - scores[2]

  refers to the value 94 (the 3rd value in the array)

- That expression represents a place to store a single integer and can be used wherever an integer variable can be used.

Declaring Arrays

The scores array could be declared as follows:

```
int[] scores = new int[10];
```

The type of the variable scores is int[] (an array of integers)

Note that the array type does not specify its size, but each object of that type has a sp
size

The reference variable scores is set to a new array object that can hold 10 integers

An array is an object, therefore all the values are initialized to default ones (here 0)

Array Example

An array element can be assigned a value, printed, or used in a calculation:

```
scores[2] = 89;

scores[first] = scores[first] + 2;

mean = (scores[0] + scores[1])/2;

System.out.println ("Top = " + scores[5]);
```

Array Example

Another examples of array declarations :

```
float[] prices = new
float[500];boolean[] flags;
flags = new boolean[20];
char[] codes = new char[1750];
```

Initializing Arrays

- An array can be initialized while it is created as fallows:

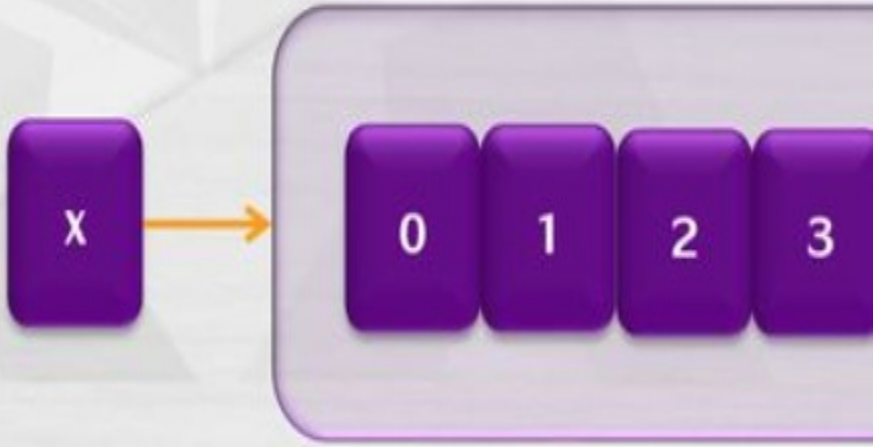
```
int [ ] x = {1, 2, 3, 4};
char [ ] c = { 'a', 'b', 'c'};
```

- To refer any element of array we use subscript or index of that location
- First location of an array always has subscript 0 (Zero)
- For example:
  - x [0] will give 1
  - c [1] will give b

Length of an Array

- Unlike C, Java checks the boundary of an array while accessing an element in it
- Programmer is not allowed to exceed its boundary
- And so, setting a for loop as follows is very common:

```
for (int i = 0; i < x.length; ++i) {
        x [i] = 5;
}
```



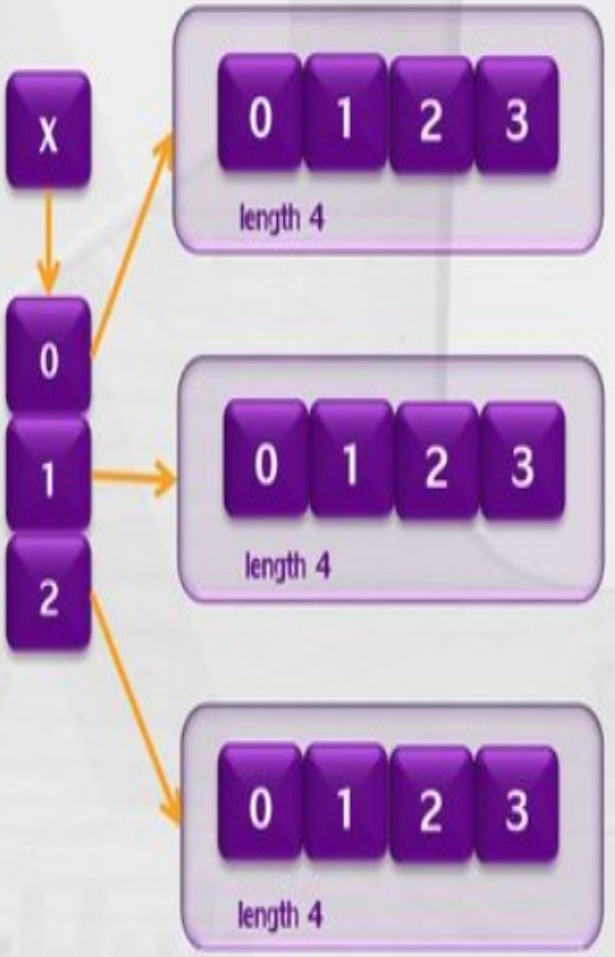
This works for **any size** array

Array Example

```
public class ArrayDemo {
        public static void main(String[ ] args) {
                int x[ ] = new int [5];
                // loop to assign the values to array
                for(int i = 0; i < x.length; ++i){
                        x[i] =1+2;
                }
                // loop to print the values of array
                for(int i = 0; i < x.length; ++i){
                        System.out.println(x[i]);
                }
        }
}
```

Multidimensional Arrays

```
int [ ][ ] x;
//x is a reference to an array of int arrays
x = new int[3][4];
//Create 3 new int arrays, each having 4 elements
//x[0] refers to the first int array, x[1] to the second and so on
//x[0][0] is the first element of the first array
//x.length will be 3
//x[0].length, x[1].length and x[2].length will be 4
```



Arrays

An **array** is a collection of variables of the same type. When you need to store a list of values, such as numbers, you can store them in an array, instead of declaring separate variables for each number.

To declare an array, you need to define the type of the elements with **square brackets**.
For example, to declare an array of integers:

```
int[ ] arr;
```

The name of the array is **arr**. The type of elements it will hold is **int**.

Now, you need to define the array's capacity, or the number of elements it will hold. To accomplish this, use the keyword **new**

```
int[ ] arr = new int[5];
```

The code above declares an array of 5 integers.
In an array, the elements are ordered and each has a specific and constant position, which is called an **index**.

To reference elements in an array, type the name of the array followed by the index position within a pair of square brackets.

Example:

```
arr[2] = 42;
```

This assigns a value of 42 to the element with 2 as its index.

Note that elements in the array are identified with **zero-based** index numbers, meaning that the first element's index is 0 rather than one. So, the maximum index of the array int[5] is 4.

KGiSL MicroCollege

Initializing Arrays

Java provides a shortcut for instantiating arrays of primitive types and strings.
If you already know what values to insert into the array, you can use an **array literal**.
Example of an array literal:

```
String[ ] myNames = { "A", "B", "C", "D"};
System.out.println(myNames[2]);
```

Place the values in a **comma-separated** list, enclosed in curly braces.
The code above automatically initializes an array containing 4 elements, and stores the provided values.

> Sometimes you might see the square brackets placed after the array name, which also works, but the preferred way is to place the brackets after the array's data type.

Array Length

You can access the length of an array (the number of elements it stores) via its **length** property.
Example:

```
int[ ] intArr = new int[5];
System.out.println(intArr.length);
```

Don't forget that in arrays, indexes start from 0. So, in the example above, the last index is 4.

KGiSL MicroCollege

Arrays

Now that we know how to set and get array elements, we can calculate the sum of all elements in an array by using loops.
The **for** loop is the most used loop when working with arrays, as we can use the **length** of the array to determine how many times to run the loop.

```
int [ ] myArr = {6, 42, 3, 7};
int sum=0;
for(int x=0; x<myArr.length; x++) {
   sum += myArr[x];
}
System.out.println(sum);

// 58
```

In the code above, we declared a variable **sum** to store the result and assigned it 0.
Then we used a **for** loop to iterate through the array, and added each element's

> The condition of the **for** loop is x<myArr.length, as the last element's index is **myArr.length-1**.

Enhanced for Loop

The **enhanced for loop** (sometimes called a "for each" loop) is used to traverse elements in arrays.
The advantages are that it eliminates the possibility of bugs and makes the code easier to read.

```
int[ ] primes = {2, 3, 5, 7};

for (int t: primes) {
   System.out.println(t);
}

/*
2
3
5
7
```

The **enhanced for loop** declares a variable of a type compatible with the elements of the array being accessed.
The variable will be available within the **for** block, and its value will be the same as the current array element.
So, on each iteration of the loop, the variable **t** will be equal to the corresponding element in the array.

Notice the **colon** after the variable in the syntax.

KGiSL MicroCollege

Multidimensional Arrays

Multidimensional arrays are array that contain other arrays. The two-dimensional array is the most basic multidimensional array. To create multidimensional arrays, place each array within its own set of curly brackets.

Example of a two-dimensional array:

```
int[ ][ ] sample = { {1, 2, 3}, {4, 5, 6} };
```

This declares an array with two arrays as its elements.
To access an element in the two-dimensional array, provide two indexes, one for the array, and another for the element inside that array.
The following example accesses the first e second array of sample.

```
int x = sample[1][0];
System.out.println(x);
```

The array's two indexes are called **row index** and **column index**.

KGiSL MicroCollege

Multidimensional Arrays

You can get and set a multidimensional array's elements using the same pair of square brackets.
Example:

```
int[ ][ ] myArr = { {1, 2, 3}, {4}, {5, 6, 7} };
myArr[0][2] = 42;
int x = myArr[1][0]; // 4
```

The above two-dimensional array contains three arrays. The first array has three elements, the second has a single element and the last of these has three elements.

In Java, you're not limited to just two-dimensional arrays. Arrays can be nested within arrays to as many levels as your program needs. All you need to declare an array with more than two dimensions, is to add as many sets of empty brackets as you need. However, these are harder to maintain.
Remember, that all array members must be of the same type.

Java String

In Java, string is basically an object that represents sequence of char values. An array of characters works same as Java string. For example:

```
char[] ch={'j','a','v','a','t','p','o','i','n','t'};
String s=new String(ch);
```

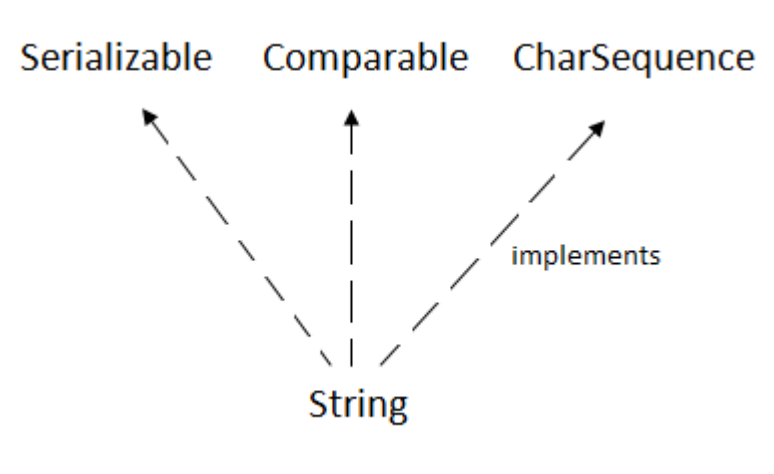
is same as:

```
String s="javatpoint";
```

Java String class provides a lot of methods to perform operations on strings such as compare(), concat(), equals(), split(), length(), replace(), compareTo(), intern(), substring() etc.

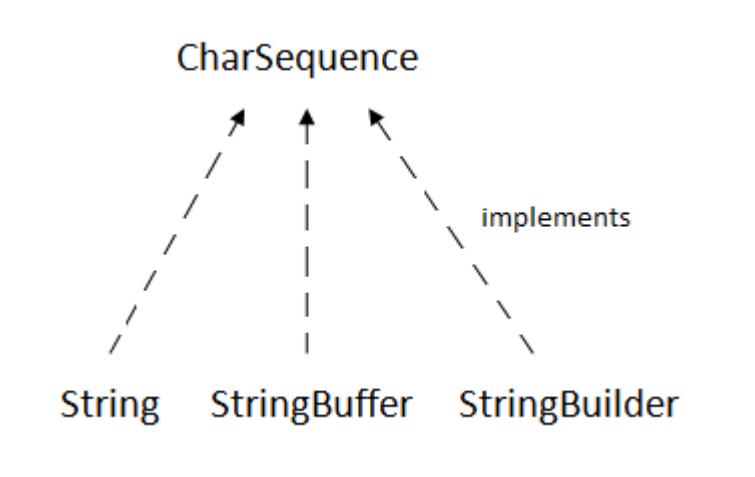
KGiSL MicroCollege

The java.lang.String class implements *Serializable*, *Comparable* and *CharSequence* interfaces.



CharSequence Interface

The CharSequence interface is used to represent the sequence of characters. String, StringBuffer and StringBuilder classes implement it. It means, we can create strings in Java by using these three classes.



The Java String is immutable which means it cannot be changed. Whenever we change any string, a new instance is created. For mutable strings, you can use StringBuffer and StringBuilder classes.

KGiSL MicroCollege

What is String in Java?

Generally, String is a sequence of characters. But in Java, string is an object that represents a sequence of characters. The java.lang.String class is used to create a string object.

How to create a string object?

There are two ways to create String object:

- By string literal
- By new keyword

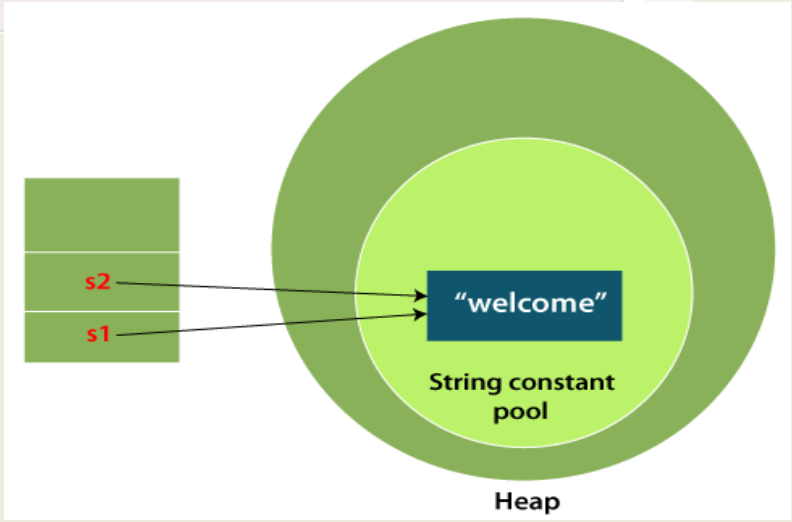
1) String Literal

Java String literal is created by using double quotes. For Example:

```
String s="welcome";
```

Each time you create a string literal, the JVM checks the "string constant pool" first. If the string already exists in the pool, a reference to the pooled instance is returned. If the string doesn't exist in the pool, a new string instance is created and placed in the pool. For example:

```
String s1="Welcome";
String s2="Welcome";//It doesn't create a new instance
```



KGiSL MicroCollege

Why Java uses the concept of String literal?

To make Java more memory efficient (because no new objects are created if it exists already in the string constant pool).

2) By new keyword

String s=new String("Welcome");//
creates two objects and one reference variable

In such case, JVM will create a new string object in normal (non-pool) heap memory, and the literal "Welcome" will be placed in the string constant pool.
The variable s will refer to the object in a heap (non-pool).

```
public class StringExample{
public static void main(String args[]){
String s1="java";//creating string by Java string literal
char ch[]={'s','t','r','i','n','g','s'};
String s2=new String(ch);//converting char array to string
String s3=new String("example");//creating Java string by new keyword
System.out.println(s1);
System.out.println(s2);
System.out.println(s3);
}}
```

KGiSL MicroCollege

Java String class methods

The java.lang.String class provides many useful methods to perform operations on sequence of char values.

No.	Method	Description
1	char charAt(int index)	It returns char value for the particular index
2	int length()	It returns string length
3	static String format(String format, Object... args)	It returns a formatted string.
4	static String format(Locale l, String format, Object... args)	It returns formatted string with given locale.
5	String substring(int beginIndex)	It returns substring for given begin index.
6	String substring(int beginIndex, int endIndex)	It returns substring for given begin index and end index.
7	boolean contains(CharSequence s)	It returns true or false after matching the sequence of char value.

KGiSL MicroCollege

Java String class methods

8	static String join(CharSequence delimiter, CharSequence... elements)	It returns a joined string.
9	static String join(CharSequence delimiter, Iterable<? extends CharSequence> elements)	It returns a joined string.
10	boolean equals(Object another)	It checks the equality of string with the given object.
11	boolean isEmpty()	It checks if string is empty.
12	String concat(String str)	It concatenates the specified string.
13	String replace(char old, char new)	It replaces all occurrences of the specified char value.
14	String replace(CharSequence old, CharSequence new)	It replaces all occurrences of the specified CharSequence.
15	static String equalsIgnoreCase(String another)	It compares another string. It doesn't check case.

Java String class methods

16	String[] split(String regex)	It returns a split string matching regex.
17	String[] split(String regex, int limit)	It returns a split string matching regex and limit.
18	String intern()	It returns an interned string.
19	int indexOf(int ch)	It returns the specified char value index.
20	int indexOf(int ch, int fromIndex)	It returns the specified char value index starting with given index.
21	int indexOf(String substring)	It returns the specified substring index.
22	int indexOf(String substring, int fromIndex)	It returns the specified substring index starting with given index.
23	String toLowerCase()	It returns a string in lowercase.

KGiSL MicroCollege

Java String class methods

24	String toLowerCase(Locale l)	It returns a string in lowercase using specified locale.
25	String toUpperCase()	It returns a string in uppercase.
26	String toUpperCase(Locale l)	It returns a string in uppercase using specified locale.
27	String trim()	It removes beginning and ending spaces of this string.
28	static String valueOf(int value)	It converts given type into string. It is an overloaded method.

KGiSL MicroCollege

Immutable String in Java

- A String is an unavoidable type of variable while writing any application program.
- String references are used to store various attributes like username, password, etc.
- In Java, **String objects are immutable**. Immutable simply means unmodifiable or unchangeable.
- Once String object is created its data or state can't be changed but a new String object is created.
- Let's try to understand the concept of immutability by the example given below:

Immutable String in Java

```
class Testimmutablestring{
 public static void main(String args[]){
   String s="Sachin";
   s.concat(" Tendulkar");//concat() method appends the string at the end
   System.out.println(s);//will print Sachin because strings are immutable objects
 }
}
```

```
Sachin
```

KGiSL MicroCollege

Immutable String in Java

```
class Testimmutablestring1{
 public static void main(String args[]){
   String s="Sachin";
   s=s.concat(" Tendulkar");
   System.out.println(s);
 }
}
```

```
Sachin Tendulkar
```

KGiSL MicroCollege

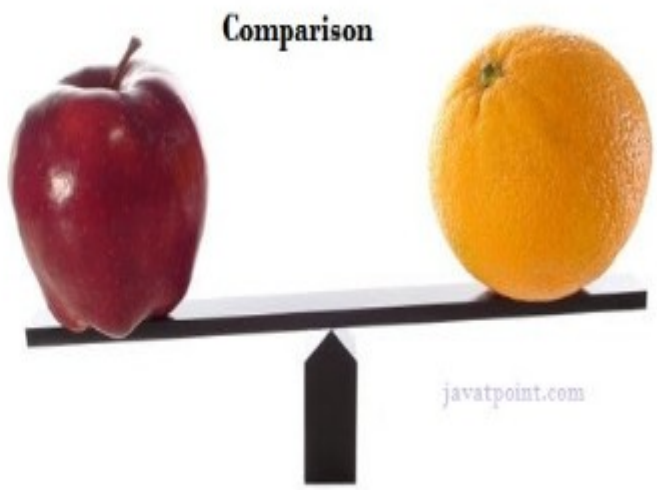
Java String compare

We can compare String in Java on the basis of content and reference.

It is used in **authentication** (by equals() method), **sorting** (by compareTo() method), **reference matching** (by == operator) etc.

There are three ways to compare String in Java:

1. By Using equals() Method
2. By Using == Operator
3. By compareTo() Method



KGiSL MicroCollege

Java String compare

By Using equals() Method

The String class equals() method compares the original content of the string. It compares values of string for equality.

String class provides the following two methods:

- **public boolean equals(Object another)** compares this string to the specified object.
- **public boolean equalsIgnoreCase(String another)** compares this string to another string, ignoring case.

Java String compare

```
class Teststringcomparison1{
 public static void main(String args[]){
   String s1="Sachin";
   String s2="Sachin";
   String s3=new String("Sachin");
   String s4="Saurav";
   System.out.println(s1.equals(s2));//true
   System.out.println(s1.equals(s3));//true
   System.out.println(s1.equals(s4));//false
 }
}
```

```
true
true
false
```

KGiSL MicroCollege

Java String compare

```
class Teststringcomparison2{
 public static void main(String args[]){
   String s1="Sachin";
   String s2="SACHIN";

   System.out.println(s1.equals(s2));//false
   System.out.println(s1.equalsIgnoreCase(s2));//true
 }
}
```

```
false
true
```

In the above program, the methods of **String** class are used. The **equals()** method returns true if String objects are matching and both strings are of same case. **equalsIgnoreCase()** returns true regardless of cases of strings.

Java String compare

By Using == operator

The == operator compares references not values.

```
class Teststringcomparison3{
 public static void main(String args[]){
   String s1="Sachin";
   String s2="Sachin";
   String s3=new String("Sachin");
   System.out.println(s1==s2);//true (because both refer to same instance)
   System.out.println(s1==s3);//false(because s3 refers to instance created in nonpool)
 }
}
```

```
true
false
```

Java String compare

By Using compareTo() method

The String class compareTo() method compares values lexicographically and returns an integer value that describes if first string is less than, equal to or greater than second string.

Suppose s1 and s2 are two String objects. If:.

s1 == s2 : The method returns 0.
s1 > s2 : The method returns a positive value.
s1 < s2 : The method returns a negative value.

Java String compare

```
class Teststringcomparison4{
 public static void main(String args[]){
   String s1="Sachin";
   String s2="Sachin";
   String s3="Ratan";
   System.out.println(s1.compareTo(s2));//0
   System.out.println(s1.compareTo(s3));//1(because s1>s3)
   System.out.println(s3.compareTo(s1));//-1(because s3 < s1 )
 }
}
```

```
0
1
-1
```

KGiSL MicroCollege

Java String compare

```
class Teststringcomparison4{
 public static void main(String args[]){
   String s1="Sachin";
   String s2="Sachin";
   String s3="Ratan";
   System.out.println(s1.compareTo(s2));//0
   System.out.println(s1.compareTo(s3));//1(because s1>s3)
   System.out.println(s3.compareTo(s1));//-1(because s3 < s1 )
 }
}
```

```
0
1
-1
```

KGiSL MicroCollege

String Concatenation in Java

In Java, String concatenation forms a new String that is the combination of multiple strings. There are two ways to concatenate strings in Java:

- **By + (String concatenation) operator**
- **By concat() method**

String Concatenation by + (String concatenation) operator

Java String concatenation operator (+) is used to add

```
class TestStringConcatenation1{
 public static void main(String args[]){
   String s="Sachin"+" Tendulkar";
   System.out.println(s);//Sachin Tendulkar
 }
}
```

```
Sachin Tendulkar
```

KGiSL MicroCollege

String Concatenation in Java

String Concatenation by concat() method

The String concat() method concatenates the specified string to the end of current string. Syntax:

public String concat(String another)

Let's see the example of String concat() method.

```
class TestStringConcatenation3{
 public static void main(String args[]){
   String s1="Sachin ";
   String s2="Tendulkar";
   String s3=s1.concat(s2);
   System.out.println(s3);//Sachin Tendulkar
  }
}
```

```
Sachin Tendulkar
```

KGiSL MicroCollege

String concatenation using StringBuilder class

- StringBuilder is class provides append() method to perform concatenation operation.
- The append() method accepts arguments of different types like Objects, StringBuilder, int, char, CharSequence, boolean, float, double.
- StringBuilder is the most popular and fastet way to concatenate strings in Java.
- It is mutable class which means values stored in StringBuilder objects can be updated or changed.

```java
public class StrBuilder
{
    /* Driver Code */
    public static void main(String args[])
    {
        StringBuilder s1 = new StringBuilder("Hello");    //String 1
        StringBuilder s2 = new StringBuilder(" World");    //String 2
        StringBuilder s = s1.append(s2);   //String 3 to store the result
            System.out.println(s.toString());  //Displays result
    }
}
```

```
Hello World
```

KGiSL MicroCollege

String concatenation using format() method

String.format() method allows to concatenate multiple

ved by the

```
public class StrFormat
{
   /* Driver Code */
   public static void main(String args[])
   {
      String s1 = new String("Hello");    //String 1
      String s2 = new String(" World");    //String 2
      String s = String.format("%s%s",s1,s2);   //String 3 to store the result
         System.out.println(s.toString());  //Displays result
   }
}
```

```
Hello World
```

KGiSL MicroCollege

String concatenation using format() method

String.format() method allows to concatenate multiple

ved by the

```
public class StrFormat
{
   /* Driver Code */
   public static void main(String args[])
   {
      String s1 = new String("Hello");    //String 1
      String s2 = new String(" World");    //String 2
      String s = String.format("%s%s",s1,s2);   //String 3 to store the result
         System.out.println(s.toString());  //Displays result
   }
}
```

```
Hello World
```

KGiSL MicroCollege

Substring in Java

A part of String is called **substring**.

In other words, substring is a subset of another String. Java String class provides the built-in substring() method that extract a substring from the given string by using the index values passed as an argument.

In case of substring() method startIndex is inclusive and endIndex is exclusive.

- You can get substring from the given String object by one of the two methods:

public String substring(int startIndex):

This method returns new String object containing the substring of the given string from specified startIndex (inclusive). The method throws an IndexOutOfBoundException when the startIndex is larger than the length of String or less than zero.

public String substring(int startIndex, int endIndex):

This method returns new String object containing the substring of the given string from specified startIndex to endIndex. The method throws an IndexOutOfBoundException when the startIndex is less than zero or startIndex is greater than endIndex or endIndex is greater than length of String.

Substring in Java

```
public class TestSubstring{
 public static void main(String args[]){
 String s="SachinTendulkar";
 System.out.println("Original String: " + s);
 System.out.println("Substring starting from index 6: " +s.substring(6));//Tendulkar
 System.out.println("Substring starting from index 0 to 6: "+s.substring(0,6)); //Sachin
 }
}
```

```
Original String: SachinTendulkar
Substring starting from index 6: Tendulkar
Substring starting from index 0 to 6: Sachin
```

KGiSL MicroCollege

Java String toUpperCase() and toLowerCase() method

The Java String toUpperCase() method converts this

tter.

```
public class Stringoperation1
{
public static void main(String ar[])
{
String s="Sachin";
System.out.println(s.toUpperCase());//SACHIN
System.out.println(s.toLowerCase());//sachin
System.out.println(s);//Sachin(no change in original)
}
}
```

```
SACHIN
sachin
Sachin
```

KGiSL MicroCollege

Java String trim() method

The String class trim() method eliminates white spaces before and after the String.

Stringoperation2.java

```
public class Stringoperation2
{
public static void main(String ar[])
{
String s="  Sachin  ";
System.out.println(s);//  Sachin
System.out.println(s.trim());//Sachin
}
}
```

KGiSL MicroCollege

Java String length() Method

The String class length() method returns length of the specified String.

Stringoperation5.java

```
public class Stringoperation5
{
public static void main(String ar[])
{
String s="Sachin";
System.out.println(s.length());//6
}
}
```

KGiSL MicroCollege

Java String length() Method

The String class length() method returns length of the specified String.

Stringoperation5.java

```
public class Stringoperation5
{
public static void main(String ar[])
{
String s="Sachin";
System.out.println(s.length());//6
}
}
```

KGiSL MicroCollege

Java String valueOf() Method

The String class valueOf() method coverts given type such as int, long, float, double, boolean, char and char array into String.

Stringoperation7.java

```
public class Stringoperation7
{
public static void main(String ar[])
{
int a=10;
String s=String.valueOf(a);
System.out.println(s+10);
}
}
```

Output:

```
1010
```

KGiSL MicroCollege

Java String charAt() Method

The String class charAt() method returns a character at specified index.

Stringoperation4.java

```
public class Stringoperation4
{
public static void main(String ar[])
{
String s="Sachin";
System.out.println(s.charAt(0));//S
System.out.println(s.charAt(3));//h
}
}
```

KGiSL MicroCollege

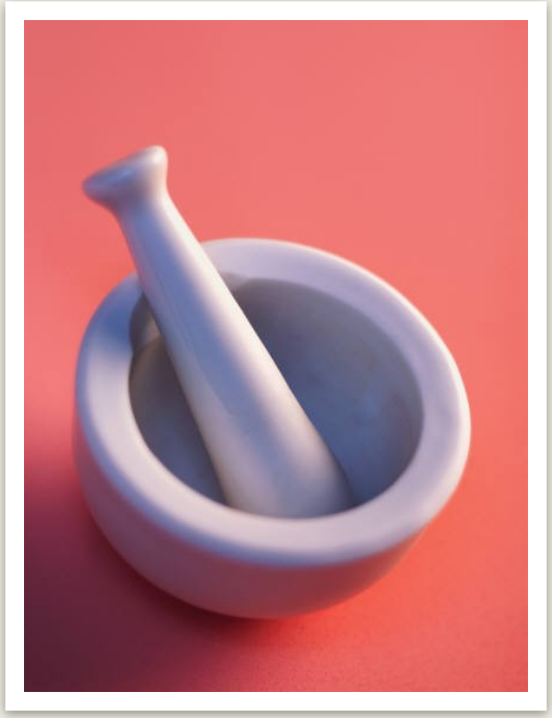
Java StringBuffer Class

Java StringBuffer class is used to create mutable (modifiable) String objects. The StringBuffer class in Java is the same as String class except it is mutable i.e. it can be changed.

Important Constructors of StringBuffer Class

Constructor	Description
StringBuffer()	It creates an empty String buffer with the initial capacity of 16.
StringBuffer(String str)	It creates a String buffer with the specified string..
StringBuffer(int capacity)	It creates an empty String buffer with the specified capacity as length.

KGiSL MicrCollege



Thank You