Course Title: Object Oriented Programming
CSBS 3252

Procedural language:

Procedural Language is also known as 3GL which means third generation language. It is a type of programming language that follows a procedure. It has a set of functions, instructions, and statements. That must be executed in a certain order to accomplish a job or program. In general, procedural language is used to specify the steps that the computer takes to solve a problem. Computer procedural languages include BASIC, C, FORTRAN, and Pascal.

Disadvantages of Procedural Language

- It is unable to solve real-world solutions.
- It is less secure because there are no data protection procedures.
- Modern features, like Encapsulation, Inheritance, Abstraction, etc. can't be implemented.
- Its semantics are tough to understand.
- It returns only restricted data types and permitted values.
- The data and functions are separate from each other.
- Global data is freely moving and is shared among various functions. Thus, it becomes difficult for programmers to identify and fix issues in a program that originate due to incorrect data handling.
- Changes in data types need to be carried out manually all over the program and in the functions using the same data type.
- Limited and difficult code reusability.

Object-Oriented Programming (OOP):

OOP is a fundamental programming paradigm based on the concept of "objects". These objects can contain data in the form of fields (often known as attributes or properties) and code in the form of procedures (often known as methods). The core concept of the object-oriented approach is to break complex problems into smaller objects.

Differences of Procedural Programming vs Object-Oriented Programming

Procedural Programming	Object-Oriented Programming
Program is divided into small parts called ***functions***.	Program is divided into small parts called ***objects***.
Follows a ***top-down approach***.	Follows a ***bottom-up approach***.
There is no access specifier	OOP has access specifiers like private, public, protected, etc.
Adding new data and functions is not easy.	Adding new data and functions is easy.
Does not have any proper way of hiding data, so it is ***less secure***.	Provides data hiding so it is ***more secure***.
Overloading is not possible.	Overloading is possible in OOP.
There is no concept of data hiding and inheritance.	The concept of data hiding and inheritance is used.
Function is more important than the data.	Data is more important than function.
Based on the ***unreal world***.	Based on the ***real world***.
PUsed for designing medium-sized programs.	Used for designing large and complex programs.
Uses the concept of procedure abstraction.	Uses the concept of data abstraction.
Code reusability absent in procedural programming,	Code reusability present in OOP.
Examples: C, FORTRAN, Pascal, Basic, etc.	**Examples:** C++, Java, Python, C#, etc.

Principles of Object-Oriented Systems:

The framework of object-oriented systems is based on the object model, which consists of two categories of elements.

a) **Major Elements** In object-oriented programming, a major element means that if a model lacks any of these components, it will not a considered object-oriented system. The four major elements are:
   - Abstraction - hiding the background details from the outside world
   - Encapsulation - wrapping up of data under a single unit
   - Polymorphism - use variables of different types at different times.
   - Inheritance - inherit properties and behaviours from another class

b) **Minor Elements** − Minor elements are those that are useful but not essential to the object model. The three minor elements are"
   - Typing
   - Concurrency
   - Persistence

Properties of object-oriented programming (OOP) language:

Object-Oriented Programming Concepts

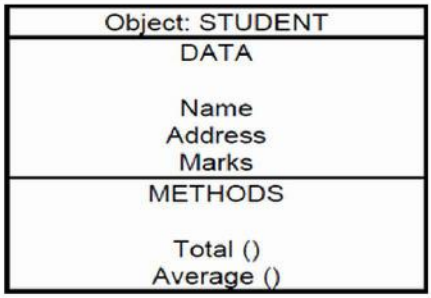
1. Object
2. Class
3. Inheritance
4. Polymorphism
5. Abstraction
6. Encapsulation

Classes and objects are the two main aspects of object-oriented programming.

Object:

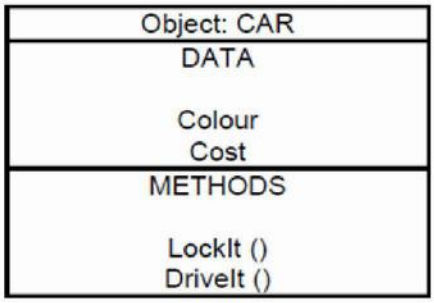
Object is a real word entity that has state and procedure is known as an object. Objects are important runtime entities in object oriented methods. An object is an entity that has two characteristics (attributes/data and method/procedure). Some of the real-world objects are books, mobile, table, computer, etc. A class has the data members (attributes) and methods, these methods and data members are accessed through an object. Thus, an object is an instance of a class.

They may characterize a location, a bank account, and a table of data or any entry that the program must handle. For example:



Representation of an object "STUDENT"

Each object holds data and code to operate the data. Objects can interact without having to identify the details of each other's data or code. Another example of object is CAR

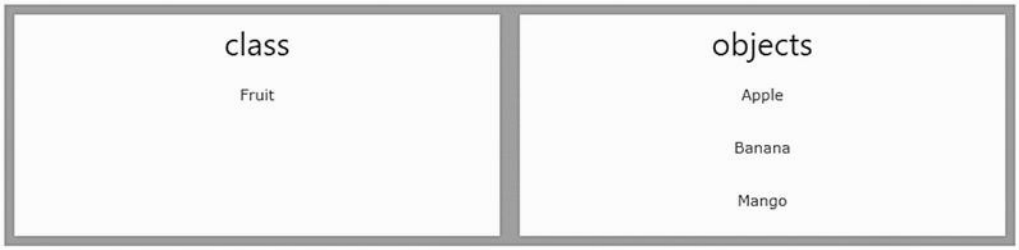


Representation of object "CAR"

Class:

An object is an instance of a class. A class is a template or blueprint from which objects are created. An object is a real-world and runtime entity. A class is a group of objects which have common properties. It is a template or blueprint from which objects are created. It is a logical entity. It can't be physical.

Example:



Properties of Java Classes:

- A class does not take any byte of memory.
- A class is just like a real-world entity, but it is not a real-world entity. It's a blueprint where we specify the functionalities.
- A class contains mainly two things: Methods and Data Members.
- A class can also be a nested class.
- Classes follow all of the rules of OOPs such as inheritance, encapsulation, abstraction, etc.

Relationships among Objects in OOAD:

In Object-Oriented Analysis and Design (OOAD), the relationship among objects is fundamental to modelling the behavior and structure of a system, and Object diagrams are used to depict instances of classes and their relationships at a specific point in time. There are several types of relationships among objects:

1. Association: The association represents a relationship between two or more objects where they are connected through a link. It can be a one-to-one, one-to-many, or many-to-many relationship. For example, a "Student" object may be associated with a "Course" object through an enrollment relationship.

2. Aggregation: Aggregation is a special form of association where a part-whole relationship exists between objects, and the part (child) can exist independently of the whole (parent). For example, a "Car" object can have an "Engine" object as a part, but the engine can exist outside the car.

3. Composition: Composition is a stronger form of aggregation where the part (child) cannot exist without the whole (parent). For example, a "House" object may be composed of "Room" objects, and if the house is destroyed, the rooms are also destroyed.

4. Inheritance: Inheritance is a mechanism where one class (subclass or child class) inherits properties and behaviours from another class (superclass or parent class). It allows for code reuse and supports the "is-a" relationship. For example, a "Dog" class can inherit from an "Animal" class.

5. Dependency: Dependency represents a relationship where one class (client) depends on another class (supplier) for its functionality. If the supplier class changes, the client class may need to be modified. For example, a "Person" class may depend on a "Car" class to drive, but the person can exist without the car.

Generalization:

Generalization in OOP is the process of identifying and combining shared characteristics of multiple classes into a single superclass. Generalization is the process of extracting common characteristics from a set of classes and abstracting them into other classes. This process helps to reduce redundancy and make code more modular and efficient.

Importance of Generalization:

- **Code Reusability:** Generalization enables the reuse of code by allowing common attributes and behaviours to be inherited by multiple classes.
- **Abstraction:** Generalization promotes abstraction by identifying commonalities among classes and abstracting them into a higher-level superclass.
- **Ease of Maintenance:** By defining common characteristics in a superclass, changes made to those characteristics propagate to all subclasses, reducing the need for redundant modifications.
- **Polymorphism:** Generalization facilitates polymorphism, allowing objects of different subclasses to be treated interchangeably through their common superclass interface.
- **Hierarchical Organization:** Generalization provides a structured way to organize classes into hierarchies, making the system easier to understand and maintain.

Meta Class:

In OOP, a Meta class can be simply defined as a type of class which is majorly utilized for the purpose of defining the functionality and behaviour of various other classes. A regular class in programming language is mainly utilized for defining the blueprint that gets used for creation of objects. But however, a Meta class is utilized for the purpose of defining how the classes are created themselves.

Differences between Meta class and Class in Java

- The features and functionalities of Meta class are found in OOP languages. But it is important to note that the features of Meta class are not supported and provided in the Java programming language.
- In Java, a class can be considered as a blueprint that is mainly utilized for defining the attributes, class variables, methods, etc. On the other hand, a Meta class can be simply defined as a type of class which is majorly utilized for the purpose of defining the functionality and behaviour of various other classes.
- Metaclass is majorly utilized for defining and controlling the behaviours of other classes. But unfortunately, there is no equivalent of metaclass found in Java programming language.
- In conclusion, classes in Java can be considered as building or fundamental blocks that are utilized for the creation of objects and they act as a blueprint. Whereas, the concepts of metaclass are not supported and provided in Java, several features such as reflection

are utilized for dynamically inspecting and manipulating classes during the runtime execution.

Message Passing:

In OOPS, message passing is a way for objects to communicate within a program. Objects communicate with one another by sending and receiving information. A message for an object is a request for the execution of a procedure and therefore will invoke a function in the receiving object that generates the desired results. Message passing involves specifying the name of the object, the name of the function, and the information to be sent.

When message passing is needed in OOP?

- When one object needs to request data from another object
- When one object needs to notify another object (or group of objects) of an event
- When multiple objects need to collaborate to achieve a common goal
- When one object needs to delegate responsibility to another object
- When one object needs to control the behavior of another object
- When one object needs to communicate with other objects in a concurrent system

Advantages of message passing:

- Message passing helps us to create modular, loosely coupled and self-contained objects that can easily interact with each other.
- Message passing enables us to design scalable systems.
- New objects can be added and existing ones can be modified without affecting the system functionality.
- Message passing enables objects to communicate asynchronously.
- This will make it easier to build concurrent systems that can handle multiple tasks simultaneously.

Disadvantages of message passing:

- Complex interactions between multiple objects can result in a complicated software design.
- Debugging can also be difficult when object interactions are not clear.
- When a large number of messages are involved, the overhead of sending and receiving messages can impact system performance.
- Overusing message passing can create tightly coupled objects.

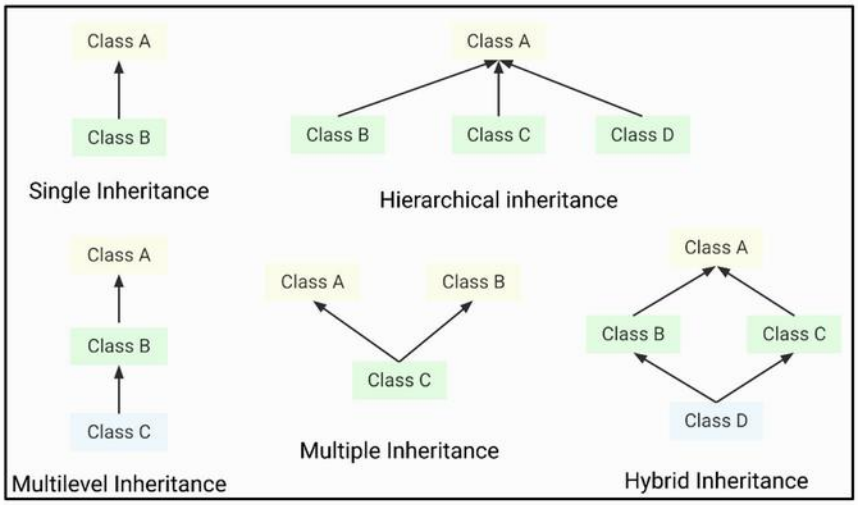
Inheritance:

Inheritance in OOP is a mechanism that allows a class to inherit properties and methods from another class. This process creates a hierarchy of classes, where the class that inherits is called the subclass or child class, and the class it inherits from is called the parent class, superclass, or base class.

Benefits of inheritance:

- **Code reuse:** Inheritance allows programmers to reuse code by referencing the behaviours and data of an existing class.
- **Simplicity:** Inheritance makes programming simpler and cleaner by avoiding the need to rewrite the same code for new classes.
- **Extensibility:** Inheritance allows programmers to extend the functionality of a base class by adding new features or overriding existing ones

Types of inheritance:



- **Single inheritance**: A child class extends from only one parent class
- **Multiple inheritance**: A child class extends from more than one parent class
- **Multilevel inheritance**: Classes with multiple levels. ***Java doesn't support this.***
- **Hierarchical inheritance**: A hierarchy of classes with a hierarchical structure
- **Hybrid inheritance**: A type of inheritance that offers a different structural relationship between classes

Encapsulation:

Encapsulation is a fundamental principle of OOP in Java. It's the process of bundling data and methods into a single unit, known as a class. This process hides sensitive data from users and prevents accidental data modification.

How it works

- Encapsulation hides a class's variables from other classes.
- Only the class's methods can access the variables.
- The class's members and methods are made private.
- Public get and set methods are provided to access and update the private variables.

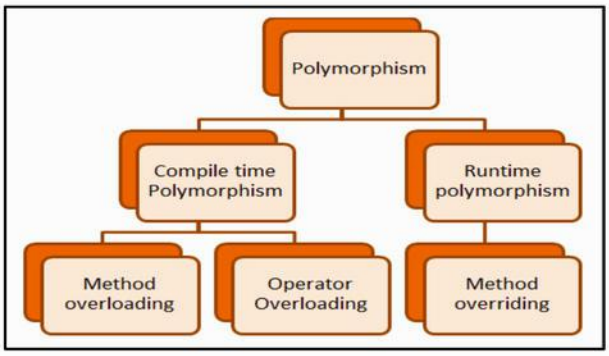
Benefits of encapsulation

- **Data hiding:** Encapsulation protects sensitive data from users.
- **Increased flexibility:** Changes to the internal implementation of a class don't affect the external code.
- **Improved maintainability:** Encapsulation keeps fields private and provides public methods to modify and view them.
- **Security:** Encapsulation protects code and data from cyber attacks.

Polymorphism:

Polymorphism in Java is a fundamental concept in object-oriented programming (OOP) that allows objects to be treated as instances of their parent class. It enables the execution of a single action in multiple ways.

Types of Polymorphism:



Polymorphism is mainly divided into two types:

- **Compile-Time Polymorphism:** In Java, this is also known as static polymorphism. This type of polymorphism is achieved by function overloading or operator overloading.
  - **Method overloading:** Defines multiple methods with the same name but different parameters within the same class.
  - **Operator overloading:** It is an idea of giving special meaning to an existing operator without changing its original meaning. ***Java doesn't support this***.
- **Runtime Polymorphism:** In Java, this is known as Dynamic Method Dispatch. It is a process in which a function call to the overridden method is resolved at Runtime.
  - **Method overriding:** A subclass provides a specific implementation of a method already defined in its superclass.

Benefits

- **Code reusability:** Allows for the reuse of code across multiple classes.
- **Flexibility:** Allows for the definition of methods in multiple forms.
- **Simplified coding:** Allows for the implementation of cleaner and more maintainable codebases.

Features of Java or Java Buzzwords:

The Java programming language can be characterized by the following buzzwords:

1. Simple
2. Object-Oriented
3. Distributed
4. Compiled and Interpreted
5. Robust
6. Secure
7. Architecture-Neutral
8. Portable
9. High Performance
10. Multithreaded
11. Dynamic

1. Simple

- Java is designed to be easy for beginners and professional programmers to learn and use effectively.
- It's simple and easy to learn if you already know the basic concepts of Object-Oriented Programming.

- Java has removed many complicated and rarely-used features, such as explicit pointers and operator overloading.

2. Object-Oriented

- Everything in Java revolves around **objects** and **classes**.
- Java allows you to model real-world entities (like a car or a bank account) as objects in your program, making it easier to manage and build complex applications.
- Key Object-Oriented Programming (OOP) concepts include:
    - **Object**: An instance of a class.
    - **Class**: A blueprint for creating objects.
    - **Inheritance**: Allows one class to inherit the properties of another.
    - **Polymorphism**: The ability of objects to take on multiple forms.
    - **Abstraction**: Hides the complex details and shows only the essentials.
    - **Encapsulation**: Keeps the data safe by restricting access to it.

3. Distributed

- Java is designed to create distributed applications on networks.
- Java applications can access remote objects on the Internet as easily as they can do in the local system.
- Java enables multiple programmers at multiple remote locations to collaborate and work together on a single project.

4. Compiled and Interpreted

- Java combines both compiled and interpreted approaches, making it a two-stage system.
- **Compiled**: Java compiles programs into an intermediate representation called Java Bytecode.
- **Interpreted**: Bytecode is then interpreted, generating machine code that can be directly executed by the machine that provides a JVM.

5. Robust

- Java provides many features that make programs execute reliably in a variety of environments.
- Java is a strictly typed language that checks code at compile time and runtime.
- Java handles memory management with garbage collection and captures serious errors through exception handling.

6. Secure

- Java does not use pointers, which helps prevent unauthorized memory access.
- The JVM verifies Java bytecode before execution, ensuring that it adheres to Java's security constraints.
- Java applications run in a restricted environment (sandbox) that limits their access to system resources and user data, enhancing security.

7. Architecture-Neutral

- Java language and JVM help achieve the goal of "write once; run anywhere, anytime, forever."
- Changes and upgrades in operating systems, processors, and system resources do not force any changes in Java programs.

8. Portable

- Java provides a way to download programs dynamically to various types of platforms connected to the Internet.

- Java is portable because of the JVM, which provides a consistent runtime environment across different platforms.

9. High Performance

- Java performance is high because of the use of bytecode.
- The bytecode can be easily translated into native machine code.

10. Multithreaded

- Multithreaded programs handle multiple tasks simultaneously, which is helpful in creating interactive, networked programs.
- Java run-time system supports multi process synchronization for constructing interactive systems.

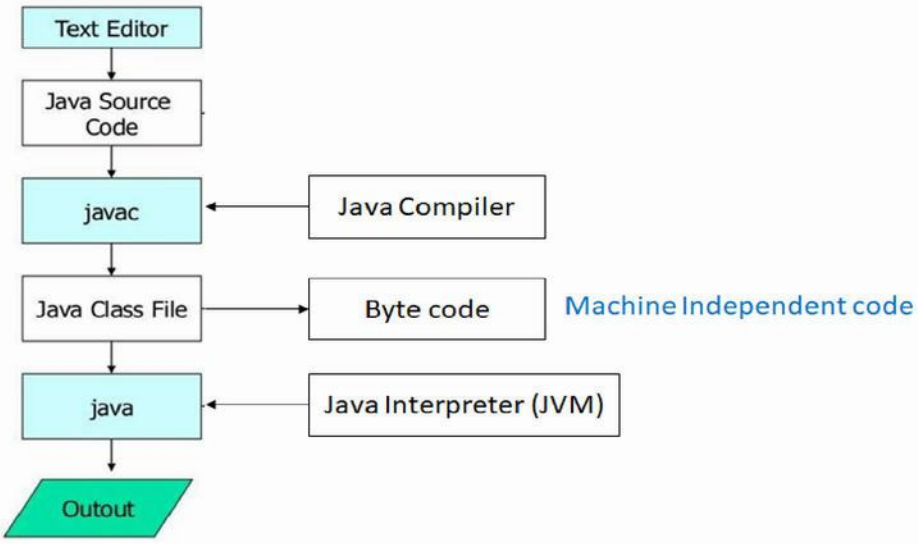
11. Dynamic

- Java can link in new class libraries, methods, and objects dynamically.
- Java programs carry substantial amounts of run-time type information, enabling dynamic linking in a safe and expedient manner.

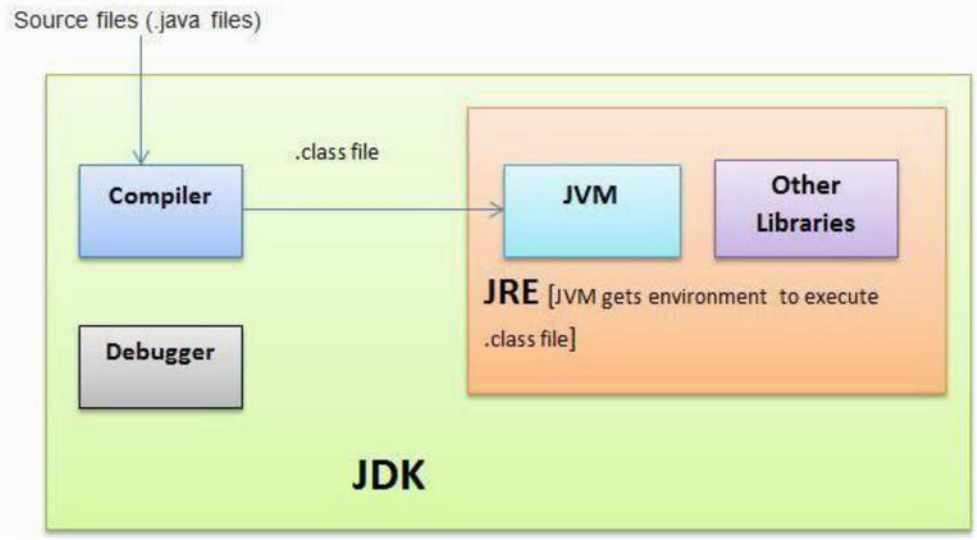
Java Applications:

- **We can develop two types of Java programs:**
  - Stand-alone applications
  - Web applications (applets)
- **Different ways to run a Java executable are:**
  - Application- A stand-alone program that can be invoked from the command line. A program that has a "main" method
    - Executed by the Java interpreter.
  - Applet- A program embedded in a web page, to be run when the page is browsed. A program that contains no "main" method
    - Java enabled web browser

Process of Building and Running Java Programs:



Above figure shows the steps of the compilation and running process of the Java program.



Above figure shows the JDK package

Java Development Kit (JDK):

JDK consists with the following below:

- javac - The Java Compiler
- java - The Java Interpreter
- jdb- The Java Debugger
- appletviewer -Tool to run the applets
- javap - to print the Java bytecodes
- javaprof - Java profiler
- javadoc - documentation generator
- javah - creates C header files

Java Compiler: Java source code (file extension .java) to bytecode (file extension .class).
Bytecode: an intermediate form, closer to machine representation.
Java Runtime Environment (JRE): It is a set of software tools that allows Java programs to run on a computer. It's a core component of the Java platform, along with the Java Virtual Machine (JVM).
Interpreter (JVM): Any target platform interprets the bytecode. Porting the java system to any new platform involves writing an interpreter. The interpreter will figure out what the equivalent machine dependent code to run.

Building your first Java Program:

```
// java source file name is : hello.java
class Hello
{
        public static void main(String args[])
        {
                System.out.println("Hello World");
        }
}
```

Command to compile the Java source code:

javac hello.java

Results: It generates the Hello.class file, This is a Bytecode File.

Command to Execute the Java class file:

java Hello

Output:

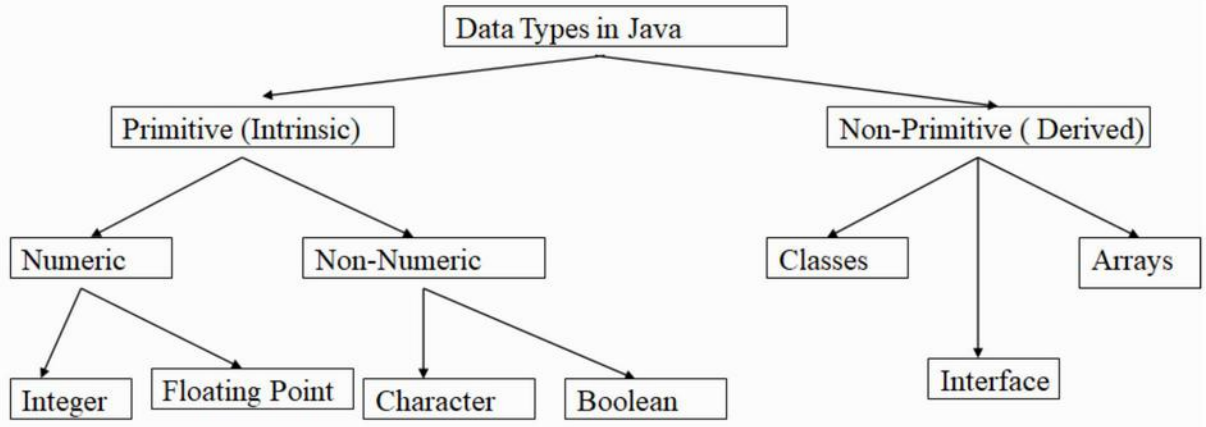
Hello World

Java main() method:

- Java Program starts executing at main() method
- main() method must be there in a class
- public, key word controls access-member may be accessed by code outside the class in which it is declared
- The main function must be public
- Keyword static allows main() to be called without having to instantiate a particular instance of class.
- main() is called by java interpreter before any object is made
- Keyword void tells the compiler main() does not return a value
- Java compiler will compile classes with main() method

Data Types:

- Java defines **eight** simple (or elemental) types of data:
  - byte, short, int, long, char, float, double, and boolean.
- These can be put in **four** groups:
  - **Integers:** This group includes byte, short, int, and long, which are for whole valued signed numbers.
  - **Floating-point numbers**: This group includes float and double, which represent numbers with fractional precision.
  - **Characters:** This group includes char, which represents symbols in a character set, like letters and numbers.
  - **Boolean:** This group includes boolean, which is a special type for representing true/false values.



Integer Types

Java consists of four integer types: byte, short, int, and long, which are defined as 8-, 16-, 32-, and 64-bit signed values as summarized in Table on next slide:

The Java integer primitive types:

Type	*Bit Size*	*Minimum Value*	*Maximum Value*
byte	8	-128	+127
short	16	-32,768	32,767
int	32	-2,147,483,648	2,147,483,647
long	64	-9,223,372,036,854,775,808	9,223,372,036,854,775,807

Integer literals can be specified in decimal, hexadecimal, or octal notation. To specify a decimal value, simply use the number as normal. To indicate that a literal value is long, you can append either "L" or "l" to the end of the number. Hexadecimal values are given in base 16 and include the digits 0-9 and the letters A-F. To specify a hexadecimal value, use 0x followed by the digits and letters that comprise the value. Similarly, an octal value is identified by a leading 0 symbol. For examples of specifying integer literals, see next Table

Examples of integer literals:

Integer	*Long*	*Octal*	*Hexadecimal*
0	0L	0	0x0
1	1L	01	0x1
10	10L	012	0xA
15	15L	017	0XF
16	16L	020	0x10
100	100L	0144	0x64

Floating-Point Types:

Support for floating-point numbers in Java is provided through two primitive types-float and double, which are 32- and 64-bit values, respectively.

Arithmetic Operators and Expressions:

An expression is an operator and operands. It follows the rules of algebra and should be familiar. Java allows several types of expressions.

Arithmetic operators:

+ addition
- subtraction
* multiplication
/ division
% modulus
+= addition assignment
- = subtraction assignment
*= multiplication assignment
/= division assignment
%= modulus assignment
++ increment
--decrement

Boolean Operators:

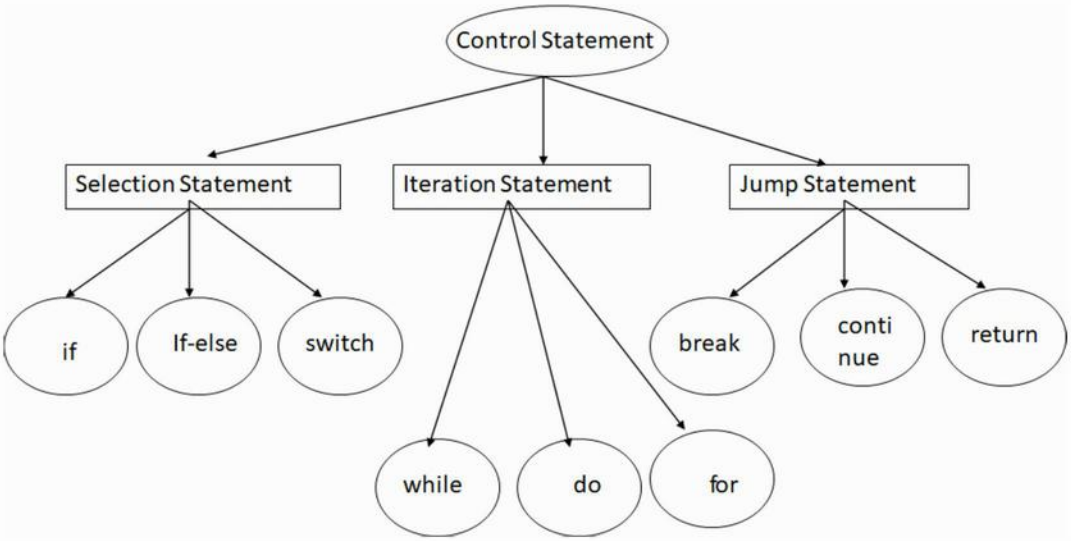
< Less than
<= Less than or equal to
> Greater than
>= Greater than or equal to

==equality
==not equality

The Conditional Operators:

Logical-AND: &&
Logical-OR: ||

Java Control Statements:



Syntax of Java Control Statements:

```
if(condition)
{
        statement;
        .......
}
else
{
        statement;
        .......
}
switch(expression)
{
case x:
        statement;
        break;
case y:
        statement;
        break;
default:
        statement;
}
```

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

while(condition)
{
        statement;
        .......
}

do
{
        statement;
        .......
} while(condition);
```

Command line Arguments in Java Program:

- It is an argument as space-separated values i.e. passed at the time of running.
- The arguments passed from the console/command prompt/terminal and it is used as an input.
- We can pass both strings and primitive data types (int, double, float, char, etc.) as command-line arguments. These arguments convert into a string array and are provided to the main() function as a string array argument.

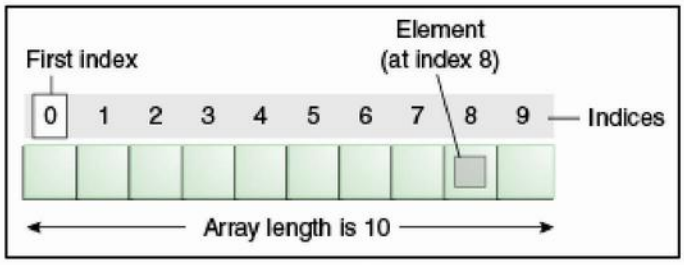
Example:

```
class MySum                    // Java Source Code  File : sum.java
{
        public static void main(String args[])
        {
                int a = Integer.parseInt(args[0]);
                int b = Integer.parseInt(args[1]);
                int sum = a + b;
                System.out.println("Result is "+sum);
        }
}
```

Command to compile the Java source code:
javac sum.java
Results: It generates the MySum.class file, This is a Bytecode File.
Command to Execute the Java class file:
java MySum 10 20
Output:
Result is 30

Array:

An *array* is a container object in Java that holds a fixed number of values of a single type. The length of an array is established when the array is created. After creation, its length is fixed. Each item in an array is called an *element*, and each element is accessed by its numerical *index*, numbering begins with 0. An array of 10 elements is shown below. The 9th element, for example, would therefore be accessed at index 8.



Declaring a Variable to Refer to an Array:

```
int[ ] anArray;   OR   int anArray[ ];          // declares an array of integers
float anArrayOfFloats[];                        // this form is discouraged
```

Creating, Initializing, and Accessing an Array:

Two ways to create an array, you need to perform two steps:

1. **Declare Array:** This creates a variable that refers to an array. But it does not actually create storage for the array.
   Syntex: datatype Arrayname[]; OR datatype[] Arrayname;
   Example : int a[]; OR int[] a;
   An array is declared with rectangular brackets [ ]

2. **Initialize for its elements :** In Java, an array is initialized by default values when the size of the array is declared with rectangular brackets [ ].
   Syntex: Arrayname = new datatype[size];
   "new" operator is used to allocate memory for the array.
   Example: a = new int[10];
   Here array of size 10, initialized with default values (0)

OR both can be used in one statement
int a[] = new int[10];

Assigning value to an array:

Syntex: Arrayname[index] = value;;
Example: a[0] = 10;
Here assigning 10 to the first element of the array.

Accessing elements of an array:

Syntex: Arrayname[index];
Example: a[0];
Here accessing the first element of the array.

Arrays are fixed length:

- Length is specified at create time
- In java, all arrays store the allocated size in a variable named "**length**".
- We can access the length of arrays as arrayName.length
  int x = students.length;
- Accessed using the index
  int x = students [1];

Arrays – Initialization at Declaration

Arrays can be initialised with specific values directly at the time of their declaration.
Syntex: DatType arrayname[] = {list of values};
Example: int[] students = {55, 69, 70, 30, 80};
This creates and initializes the array of integers of length 5. In this case it is not necessary to use the new operator.

Arrays – Example:

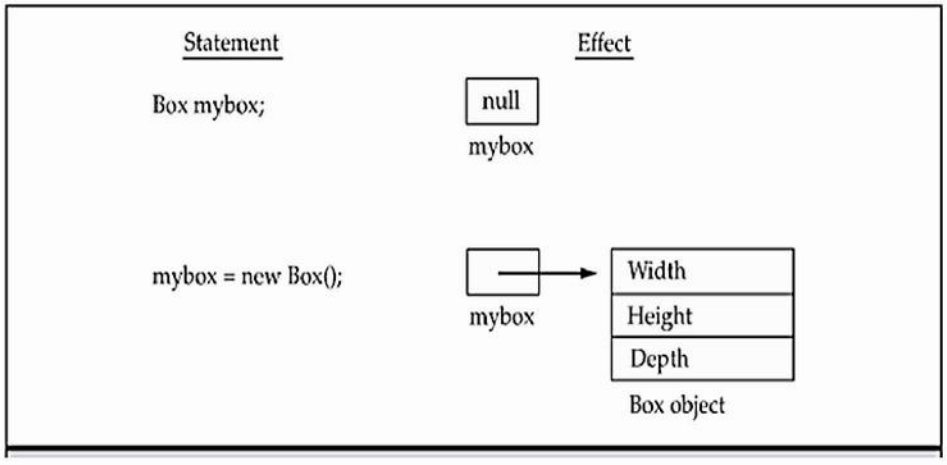
```
public class StudentArray
{
    public static void main(String[] args)
    {
        int[] students;
        students = new int[7];
        System.out.println("Array Length = " + students.length);
        for ( int i=0; i < students.length; i++)
            students[i] = 2*i;
            System.out.println("Values Stored in Array:");
        for ( int i=0; i < students.length; i++)
            System.out.println(students[i]);
    }
}
```

Create Objects and Classes in Java:

- Java is a true OO language, so the underlying structure of all Java programs is classes.
- Anything represented in Java, it must be encapsulated in a class. That defines the state and behavior of the basic program components known as objects.
- Classes create objects and objects use methods to communicate between them.
- They provide a convenient method for packaging a group of logically related data items and functions that work on them.
- A class essentially serves as a template for an object and behaves like a basic data type.
- It is important to understand how the fields and methods are defined in a class and how they are used to build a Java program that incorporates the basic OO concepts such as encapsulation, inheritance, and polymorphism.

A program that uses the Box class. BoxDemo.java

```
class Box                       //Create a class named "Box" with three attributes
{
        double width;
        double height;
        double depth;
}
class BoxDemo
{
        public static void main(String args[])
        {
                Box mybox ;                     // declare reference to object
                mybox = new Box();              // allocate a Box object
                double vol;
                mybox.width = 10;               // assign values to mybox's instance variables
                mybox.height = 20;
                mybox.depth = 15;
                vol = mybox.width * mybox.height * mybox.depth; // compute volume of box
                System.out.println("Volume is " + vol);
        }
}
```



Above figure shows the effects of the statement for creating an object.

Adding a Method to the Box Class:

```
class Box                       //Create a class named "Box" with three attributes
{
        double width;
        double height;
        double depth;
        double volume()         // display volume of a box
        {
                return width * height * depth;
        }
}
class BoxDemo
{
        public static void main(String args[])
        {
                Box mybox = new Box();      // declare and  allocate Box object
                double vol;
                mybox.width = 10;           // assign values to mybox's instance variables
                mybox.height = 20;
                mybox.depth = 15;
                vol = mybox1.volume();       // compute volume of box using method
                System.out.println("Volume is " + vol);
        }
}
```

Adding a Method That Takes Parameters:

```
class Box                       //Create a class named "Box" with three attributes
{
        double width;
        double height;
        double depth;
        double volume()                                 // display volume of a box
        {
                return width * height * depth;
        }
        void setDim(double w, double h,  double d)      //Method with parameter
        {
                width = w;  height = h;  depth = d;
        }
}

class BoxDemo
{
        public static void main(String args[])
        {
                Box mybox = new Box();              // Box object
                double vol;
                mybox.setDim(10, 20, 15);  //Passed parameters to the method
                vol = mybox1.volume();       // compute volume of box using method
                System.out.println("Volume is " + vol);
        }
}
```

Constructor:

It is a special method that is used to initialize objects. The constructor is called when an object of a class is created. It can be used to set initial values for object attributes.

- The constructor name must match the class name
- It cannot have a return type
- The constructor is called automatically when the object is created
- A Java constructor cannot be abstract, static, final, and synchronized

Types of Constructor:

1. **Default Constructor (**no-arg constructor**) :** A constructor is called "Default Constructor" when it does not have any parameter. The default constructor is used to provide the default values to the object like 0, null, etc., depending on the type.

   Syntax:
   <class_name>(){
   }

2. **Parameterized Constructor:** A constructor which has a specific number of parameters is called a parameterized constructor. It is used to provide different values to distinct objects.

   Syntax:
   <class_name>(<data_type Parameter_name>,<...>,...){
   }

   Example:

```
class Box                       //Create a class named “Box” with three attributes
{
        double width;
        double height;
        double depth;
        double volume()         // display volume of a box
        {
                return width * height * depth;
        }
        Box()                                   //Define a Default Constructor
        {
                System.out.println("Default Constructor");
                width = 10;
                height = 10;
                depth = 10;
        }
        Box(double w, double h,  double d)    //Define a parameterized Constructor
        {
                System.out.println("Parameterized Constructor");
                width = w;
                height = h;
                depth = d;
        }
```

OR USING this KEYWORD

```
        Box(double width, double height,  double depth)    //Using this keyword
        {
                System.out.println("Parameterized Constructor");
                this.width = width;
                this.height = height;
                this.depth = depth;
        }
}

class BoxDemo
{
        public static void main(String args[])
        {
                Box mybox1 = new Box();              // For Default constructor
                Box mybox2 = new Box(5,9,3);         // For Parameterized constructor
                double vol1 = mybox1.volume();       // compute for Default
                double vol2 = mybox2.volume();       // compute for Parameterized
                System.out.println("Default volume is " + vol1);
                System.out.println("Parameterized volume is " + vol2);
        }
}
```

Finalize:

- The finalize() in Java is a method of the Object class used to perform cleanup activity before destroying any object.
- Garbage collector calls it before destroying the objects from memory.
- It releases all system resources before the garbage collector runs for a specific object
- This method in Java is called by default for every object before its deletion.

Garbage Collector:

- It is an automatic memory management process that helps Java programs run efficiently.
- It identifies which objects are still in use (referenced) and which are not in use (unreferenced).
- Unreferenced objects can be deleted to free up memory.
- It runs in the background without requiring the programmer's constant attention.