Java (Latest Version Java 22)

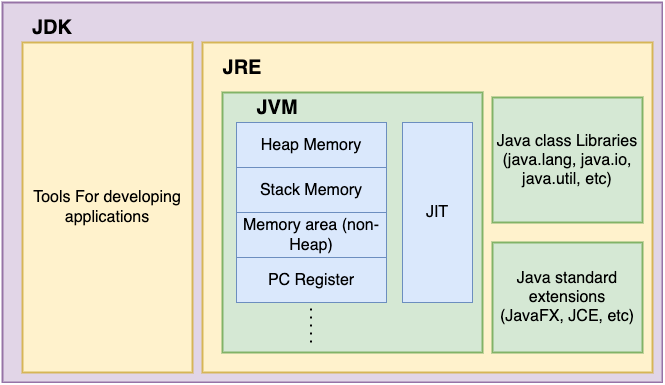
* Advantages of Java

1. Object-Oriented: Java is an object-oriented language, which means it models real-world objects and encourages the use of reusable code through classes and objects.
2. Platform Independent: Java applications are compiled into bytecode, which can be run on any system with a Java Virtual Machine (JVM). This makes Java platform-independent at the source level.
3. Simple: Java is designed to be easy to learn and use, with a syntax similar to C++, but with simplified features and no complex aspects like pointers and operator overloading.
4. Multithreaded: Java has built-in support for multithreading, allowing the development of programs that can perform multiple tasks simultaneously.
5. Garbage Collection: Java automatically manages memory allocation and deallocation, reducing the risk of memory leaks and freeing developers from manual memory management.
6. Rich Standard Library: Java provides a vast standard library (Java API) that supports many functionalities such as data structures, networking, file I/O, utilities, and more.
7. Compile And Interpreted :-

* Difference Between C++ and Java

|  |  |  |
| --- | --- | --- |
| Feature | C++ | Java |
| Programming Paradigm | Procedural and Object-Oriented | Object-Oriented |
| Platform Dependence | Compiled to machine code, platform dependent | Compiled to bytecode, platform independent (runs on JVM) |
| Memory Management | Manual | Automatic (garbage collection) |
| Pointers | Supported | Not supported |
| Pass-by-Value vs Pass-by-Reference | Supports both | Only pass-by-value |
| Primitive Data Types | int, float, double, char, etc. | Similar to C++ |
| Object-Oriented Features | Classes, inheritance, polymorphism | Similar to C++, but with stricter enforcement |
| Input/Output | cin, cout | System.in, System.out |
| Threading | Requires third-party libraries | Built-in support |
| Default Arguments | Supported | Not supported |
| Header Files | Required for function and class declarations | Not used, classes are in separate files |

* JDK, JRE, JVM, GC, Bytecode



* JDK (Java Development Kit) :
* It is a software development environment which is used to develop Java application and applets.
* It contains JRE + JDK tools.
* Tools : Javac , Java , JavaDoc , Jdb.
* JRE (Java Runtime Environment) :
* It is a set of software Tools which are used for executing Java Application. It is used to provide Runtime environment.
* It is the implementation of JVM.
* It contains set of libraries plus other files that JVM uses at run-time.
* JVM (Java Virtual Machine):
* It is a specification that provides run-time environment in which Java byte code can be executed.
* JVM's are available for many hardware and software platform.
* The JVM performs following operations. (1) Loads the Java Code (2) Verifies code (3) Executes code
* Working Flow
* The Java compiler is used to compile the Java source code. The compiler translates the .java files into bytecode, which is stored in .class files. Bytecode is a platform-independent, intermediate representation of the code. The Java Virtual Machine (JVM) loads the compiled .**class** files. The JVM interprets the bytecode using the interpreter or Just-In-Time (JIT) compiler. **The interpreter translates bytecode into machine code one instruction at a time**, executing it immediately. The JVM's execution engine manages these processes, executing the compiled code. The JVM provides a runtime environment for the Java application, handling tasks such as memory management, garbage collection, and thread management.
* Data Types

|  |  |
| --- | --- |
| Data Type | Size |
| byte | 1 byte |
| short | 2 byte |
| int | 4 byte |
| long | 8 byte |
| float | 4 byte |
| double | 8 byte |
| char | 2 byte |
| **boolean** | **1 bit** |

Role of Compiler and Interpreter in Program execution

The Java compiler (javac) takes the human-readable source code written in Java and translates it into bytecode. Bytecode is a set of instructions that is highly optimized for the Java Virtual Machine (JVM).

The interpreter executes the bytecode line-by-line. It translates the bytecode into machine code (native code) that the computer's processor can understand.

* Why char of Java takes 2 bytes of memory as compared to char in C++.
* C++ uses the ASCII code character sequence 1 byte ( 7 bit ) that means 128 characters , which is enough to represent 128 characters. But the java uses UNICODE character sequence 2 bytes ( 16 bit ).
* Java was designed with internationalization in mind, aiming to support characters from various languages and scripts around the world.
* Unicode provides a universal character set that includes characters from most of the world's writing systems.
* By using 16-bit char types, Java can natively represent a large number of characters from the Unicode standard without additional encoding schemes.
* Unicode includes over 65,000 characters in the BMP, which can be directly represented with 16 bits.
* Type Casting in Java ( IMP )
* Type casting is nothing but assigning a value of one primitive data type to another.
* The cast is performed by placing the desired type in parenthesis to the left of the value to be converted.
* Ex : int n = (int) 23.135;

Rules in type casting Source

* If source and destination of the same type. No casting is required.
* If source is of smaller size than destination then no casting is required. e.g. int to long or float to double
* If destination is of smaller size than source then casting is required. e.g. long to int or double to float.

1. Implicit Typecasting ( Widening ) :- Automatically converts a smaller data type to a larger data type. No data loss occurs.

int num = 10;

long longNum = num; // Implicit casting from int to long

1. Explicit Typecasting ( Narrowing ) :- Manually converts a larger data type to a smaller data type. May result in data loss.

double doubleNum = 9.78;

int intNum = (int) doubleNum; // Explicit casting from double to int.

* String in Java
* Strings are sequence of characters or collection of characters.
* Java does not have a built-in string type instead the standard java library contains a pre-defined class called "String"
* "String" class is declared in "java.lang" package.
* Java string is immutable (not changeable).
* Methods of String Class
  + length ( ) : - This method is used to find the length of the string
  + charAt ( ) :- This method returns the character from specified index of the string.
  + indexOf( ) :- This method returns an index of first occurrences of given character/sub-string in the string. If not present in the string, then it returns -1.
  + lastIndexOf ( ) : - Returns the index within the string of the last occurrence of the specified character.
  + equals ( ) This method compares two strings for equality. Returns True, if strings are equal, otherwise returns false.
  + equalsIgnoreCase ( ) This method compares two strings for equality, ignoring case consideration.
  + compareTo ( ) This method compares two strings lexicographically (character by character) the comparison is based on the Unicode value of each character in the string. (i) This method returns < 0, if first string is smaller than second. (ii) It returns > 0, if first string is greater than second string. (iii) It returns 0 , if both strings are equal.
  + endsWith ( ) Test if the string ends with specified string.
  + startsWith ( ) :- This method test if the string starts with specified string.
  + toLowerCase ( ) Converts all the Upper Case letters of the string in to lower case
  + toUpperCase( ) Converts all the lowerCase letters of the string into Uppercase.
  + trim( ) Removes any leading and trailing spaces from String.
  + substring ( ) This method returns a string that is a substring of this string. ( last character is excluding )
* How String are Immutable in Java

public static void main(String[] args) {

String str1 = "Hello";

String str2 = str1; // str2 points to the same object as str1

str1 = str1 + " World"; // Concatenation creates a new String object

System.out.println("str1: " + str1); // Outputs: str1: Hello World

System.out.println("str2: " + str2); // Outputs: str2: Hello

}

* String str1 = "Hello"; This creates a String object with the value "Hello" and stores it in memory
* String str2 = str1; This makes str2 refer to the same memory location as str1.
* str1 = str1 + " World"; This statement concatenates " World" to str1. However, because strings are immutable, Java does not modify the original str1 object. Instead, it creates a new String object with the value "Hello World" and assigns it to str1
* The original "Hello" string remains unchanged in memory, and str2 still points to it.
* StringBuffer Class in Java.
* Java provides the String and StringBuffer classes to handle String.
* The String class is used to manipulate character strings that cannot be changed. The string which cannot be changed is called as immutable string.
* The StringBuffer class is used to represent character String that can be changed after it is created. The string which can be changed is called as **mutable string**.
* StringBuffer represents growable and writable character sequence.
* Methods of StingBuffer() : -

1. append() This method will concatenate the string representation of any type of data to the end of the StringBuffer object. append() method has several overloaded forms.

Syntax: StringBuffer append(T value)

1. insert() This method inserts one string into another.

Syntax: StringBuffer insert(int index, T value)

1. delete () This method removes the characters from the string.

Syntax: StringBuffer delete (int start, int end)

This method is used to delete the string from specified startIndex and endIndex. Here endindex is exclusive.

1. deleteCharAt() This method removes the character at the specified index character from the string. Syntax: StringBuffer deleteCharAt (int index)
2. replace () This method replaces the specified string from specified start index to the end index.

Syntax: StringBuffer replace (int start, int end, String newString)

1. reverse() This method reverses the characters within a StringBuffer object.

Syntax: StringBuffer reverse ()

* Array in Java

• An array is a data structure that stores a collection of values of the same type.

• You can access each individual value through an integer index.

• The first element of array is at 0th index, second is at 1st index and so on

* Array initialization

1. Implicit

* In Java, the default values for any numeric array is zero.
* For character array, default value is null ('\0').
* For boolean array, default value is false.
* For float and double it is 0.0
* int [] a = new int[5];

1. explicit

* You can initialize elements of an array explicitly.
* The initial values for elements are separated by a comma and enclosed in braces {}.
* // Initialize the array at the time of declaration
* int[] empId = {1, 2, 3, 4, 5};
* The code above creates an array of int of length 5, and initializes its elements to 1, 2, 3, 4, and 5.
* The length of an array is the same as the number of values specified in the array initialization list.

1. Bounds Checking in Array

* When your program is running and it tries to access an element of an array, the Java virtual machine checks that the array element actually exists. This is called bounds checking.
* If your program tries to access an array element that does not exist, the Java virtual machine will generate an: **ArrayIndexOutOfBoundsException**
* For Example: Output: int [] a ={10,20,30,40,50};

System.out.println(a[8]);

ArrayIndexOutOfBoundsException

1. Arrays Class in Java
   * This class contain various methods for manipulating arrays such as sorting and searching. This class declared in java.util package.
   * Methods of Arrays Class
2. binarySearch ():

Syntax : static int binarySearch (int [ ] a, int key)

This method searches the specified array of integers for the specified value using binary search algorithm. It returns an index of the key value if it returns an index of the key value if it present, otherwise it returns negative number.

1. equals ( )

Syntax : static boolean equals (int [ ] a1, int [ ]a2)

This method returns true if the two specified arrays of integers are equal to one another.

1. fill ( )

Syntax : static void fill (int [ ], int val)

This method assigns the specified int value to each element of the specified array of integers.

1. sort ( )

Syntax : static void sort (int [ ] a)

This method sorts the specified array into ascending numerical order. It uses quick sort algorithm.

1. toString ( )

Syntax : static String toString (int [ ] a)

This method returns a string representation of the contents of the specified array.

* Array of References / Objects
* If you want to store a single object in your program, then you can do so with the help of a variable of type object. But when you are dealing with numerous objects, then it is advisable to use an array of objects.
* Actually it is not the object itself that is stored in the array but the references of the object.
* The following statement creates an Array of Objects.

Class\_name [] objArray;

* In the case of an array of objects, each element of array i.e. an object needs to be initialized.

import java.util.\*;

class EmployeeP{

int id;

String name;

int salary;

void getData(){

Scanner in = new Scanner(System.in);

System.out.println("Enter Empid, Empname and EmpSalary ");

id = in.nextInt();

name = in.next();

salary = in.nextInt();

}

void putData(){

System.out.println("\n");

System.out.println("Id : - "+id + " Name "+name + " Salary "+salary );

}

public static void main(String [] arsg)throws Exception{

Scanner in = new Scanner(System.in);

System.out.println("How many Employees ? ");

int count = in.nextInt();

EmployeeP [] emp = new EmployeeP[count]; /// **Array of References is Created**

for(int i=0;i<emp.length;i++){

emp[i] = new EmployeeP(); // **for each reference object is created (if not written throw NullPointerException)**

emp[i].getData();

}

for(int i=0;i<emp.length;i++){

emp[i].putData();

}

}

}

* Command Line Argument
* Every Java program has a main method with String [ ] args parameter.
* This parameter indicates that the main method receives an array of strings specified on the command line. Command line arguments are parameters that are supplied to the program at the time of involving it for execution.
* Any argument provided on command line passed to the array args as its element.
* We can write the program that can receive and use the arguments provided on the command line.
* Program

import java.util.\*;

class CMDP{

public static void main(String [] args){

int x = Integer.parseInt(args[0]);

int y = Integer.parseInt(args[1]);

System.out.println(x+y);

}

}

* Classes And Objects in Java
* **A class** is a user defined blueprint or prototype from which objects are created. It represents the set of properties or methods that are common to all objects of one type.
* The data in an object are called its instance fields, and the functions and procedures that operate on the data are called its methods.
* Special methods called Constructor is used to initialize the Object.
* **An Object** is an instance of a class. When you create an object from a class, you are creating a specific instance with its own set of values for the fields defined in the class. Objects can interact with each other by calling methods.
* Static Field And Static Method
* Static Field
* Sometimes there may be a situation when objects of a particular class may share some common item of data. In such situation static field is use.
* A field of class which share common value among all objects, then it declared as static field.
* Exmaple

Class Circle {

Int radius; --- 4 byte

Static Double PI; ---- 8 byte ( Only one Time )

}

* If we create the 3 objects of this Circle class and if PI is not static then we need 12 \* 3 == 36 bytes of memory.
* But if we declare the Pi as static then only one time memory is allocated to PI. i.e 8. So , memory required (12 + 8 ) 20 bytes. Here the difference of 16 bytes.
* In this case we take example of only 3 objects , but imagine if the objects are 100 or even 200.
* Initial value of static field is 0.

class StaticP{

int x;

static int y;

StaticP(int a,int b){

x = a;

y = b;

}

static void Show(){

System.out.println(" Y :- "+y);

}

public static void main(String [] args){

StaticP.Show();

StaticP p1 = new StaticP(10,20); // Y becomes 20

StaticP p2 = new StaticP(20,40); // Y becomes 40

StaticP p3 = new StaticP(30,60); // Y becomes 60

StaticP.Show();

}}

* Static Method
* A method of class can be declared as static method by preceding it’s declaration with keyword "static".
* Static method can access only other static members of class.
* Static method always called through class name.
* The main method in Java, which is the entry point of any Java application, is declared as public static. The static keyword ensures that the Java Virtual Machine (JVM) can call this method without creating an instance of the class. The public modifier allows the JVM to access it from anywhere.
* No, static methods in Java cannot be overridden because static methods are associated with the class itself rather than an instance of the class. When a subclass inherits a static method from its parent class, it is not possible to override the behaviour of that static method. They cannot be overridden because they do not act on a specific instance of an object.
* Program :-

import java.util.\*;

class StaticMethod{

static int x = 10;

int non\_static = 100;

static void AccessStaticVars(){

System.out.println(x);

System.out.println(non\_static); // Cannot Accessible

}

public static void main(String [] args){

StaticMethod mtd = new StaticMethod();

AccessStaticVars(); // 10

StaticMethod.AccessStaticVars(); // 10

}

}

* Finalize () Method
* Java does automatic garbage collection. So, manual memory de-allocation is not needed and therefore, java does not support destructors.
* Of course, some object utilize a resource other than memory, such as file or a graphic resource, network connection, database connection.
* In this case, it is important that the resource must be recycled when it is no longer needed. For this purpose, we can add finalize ( ) method to any class.
* The finalize method will be called before the garbage collector, destroy the object from the memory.
* Program :-

import java.util.\*;

class FinalizeMtd {

FinalizeMtd(){

System.out.println("Inside Constructor");

}

protected void finalize(){

System.out.println("Called Gargbage Collector Before Destroying the Object");

}

public static void main(String [] args){

FinalizeMtd f = new FinalizeMtd();

f = null;

System.out.println("GC Call");

System.gc();

}

* Why Finalize() method is defined as Protected

1. Declaring the method as protected ensures that it can only be called by the class itself, subclasses, or classes in the same package.
2. the language designers intended to give **the class control over its own cleanup process**. The garbage collector is the only entity that should call finalize(), as it's responsible for managing object lifecycle and cleanup. Allowing external code to call finalize() could lead to inconsistent states or resource leaks.
3. The finalize() method is intended to be called by the garbage collector, not manually from your code.

* Inheritance in Java
* Creating a new class from existing class is referred as Inheritance.
* When one class acquires the properties of another class then it is referred as inheritance.
* New class which is created from existing class is referred as subclass.
* Existing class from which new class is created is referred as super class.
* In OOP, concept of inheritance provides the idea of Reusability of code.
* "**extends**" keyword is used to perform the inheritance in Java.
* Types Of Inheritance
  1. Single Inheritance
  2. Multi - level Inheritance
  3. Hierarchical Inheritance
* Program

import java.util.\*;

class Sample {

int x = 10;

Sample(){

System.out.println("Inside Sample Class");

System.out.println(x);

}}

class InheritanceP extends Sample{

InheritanceP(){

System.out.println("Inside InheritanceP Class");

}

public static void main(String [] args){

InheritanceP p = new InheritanceP();

}

}

O/P ; - Inside Sample Class

X :- 10

Inside InheritanceP Class

* Super Keyword
* Whenever a subclass needs to refer to its immediate super class, it can do so by use of the keyword "**super**".
* There are two uses of Super.
  + - 1. Using super to call super class constructor
  + Whenever we created the object of sub-class , first the super class Constructor is called and then sub-class constructor is called.
  + But if super class constructor is Parameterized constructor then it is responsibility of subclass to provide the values for super class constructor.
  + So in that case from subclass to call the constructor or super class we have to use Super() keyword in subclass constructor
  + The call to super class constructor super() must be the first statement within the subclass constructor.
  + The parameters in the super call must match the order and type of the variable declared in the super class constructor.
  + Program :-

class Sample{

Sample(int a,int b){

System.out.println("A :- "+a);

System.out.println("B :- "+b);

}

}

class SuperP extends Sample{

SuperP(int val){

super(val,100);

System.out.println("Val :- " + val);

}

public static void main(String [] args){

SuperP sp = new SuperP(10);

}

}

1. Using super to access hidden member of super class in subclass.

* It may be possible that sub class have same field name as the super class field. So in that case how we distinguish or how we refer to the super class field. Because in inheritance all fields and methods are inherited in subclass also.
* In such case whenever we need to refer **field with same name** which is belongs to super class , we use super keyword
* Program : -

import java.util.\*;

class Sample{

int x;

}

class SuperP extends Sample{

int x;

SuperP(int a,int b){

super.x = a;

x = b;

}

void Show(){

System.out.println("Super Class X :- "+super.x);

System.out.println("Sub Class X :- "+x);

}

public static void main(String [] args){

SuperP sp = new SuperP(10,20);

sp.Show();

}

}

O/P :- Super Class X :- 10

Sub Class X :- 20

* Method Overloading And Overriding
  + - 1. Method Overloading
  + Overloading occurs if several methods have the same name but different

Parameters types or no. of parameters and return type.

* + The compiler call the correct method by matching the parameter types of

the various methods with the types of values used in the specific method

call.

* + A Compile Time error occurs if the compiler cannot match the parameter.

1. Method Overriding
   * In Inheritance, when a method in a subclass has the same name and same parameters list as a method in it super class, then the method in the subclass is said to override the method in the super class.
   * Program :-

class sample{

void show(){

System.out.println("Inside Sample Class");

}

}

class OverridingP extends sample{

void show(){

System.out.println("Inside OverridingP Class");

}

public static void main(String [] args){

OverridingP p = new OverridingP();

p.show();

}

}

O/P- Inside OverridingP Class

|  |  |  |
| --- | --- | --- |
| Feature | Method Overloading | Method Overriding |
| Performed within | Class | Subclass |
| Parameter List | Must be different | Must be same |
| Polymorphism Type | Compile-time polymorphism | Run-time polymorphism |

* Dynamic Method Dispatch
* DMD is the mechanism by which a call to an overridden method is resolved at runtime rather than compile time.
* DMD is important because this is how JAVA implements runtime polymorphism.
* In Java, a super class reference variable can refer to a subclass object, but it can access only those members of subclass which are inherited form super class.
* When an overridden method is called through a super class reference, Java determines which version of that method to execute, based upon the type of object being referred by the super class reference.
* This determination is made at runtime. Therefore, if a super class contains a method, that is overridden by a subclass then different types of object are referred through a super class reference, different versions of the subclass are executed.
* Program :-

import java.util.\*;

class Animal{

void travel(){

System.out.println("Animal Travel from one location to Another");

}

}

class Cow extends Animal{

void travel(){

System.out.println("Cow Travel on Earth");

}

}

class Parrot extends Animal{

void travel(){

System.out.println("Parro Travel in Sky");

}

}

class DMDP{

public static void main(String [] args){

Animal a = new Cow();

a.travel(); // Cow Travel on Earth

}}

* Abstract Method and Abstract Class
  + - 1. Abstract Method
* In Inheritance, we have some methods which must be overridden by the subclass.
* In this case, we want some way to ensure that a subclass must override all necessary methods.
* Java's solution to this problem is an Abstract method.
* We can require that certain methods must be override in subclass by specifying the abstract method in the super class
* Properties of Abstract Method
  + - Abstract method doesn't have method body ({ })
    - It is also called as unimplemented method.
    - It must be override in subclass.
      1. Abstract Class
* A class which can not be instantiated is called as **Abstract Class** ( Means object of abstract class is not created)
* Any class that contains one or more abstract method must also be declared as an abstract.
* To declare abstract class use keyword "**abstract**" in front of class declaration.
* We can't create an object of an abstract class but we can create the reference.
* Abstract class is also referred as "partially unimplemented class".

1. Program

abstract class Shapee{

abstract void area();

}

class Rect extends Shapee{

int len,bre;

Rect(int a,int b){

len = a;

bre = b;

}

void area(){

System.out.println("Area of Rectangle "+(len \* bre));

}

}

class Circle extends Shapee{

int r;

Circle(int a){

r = a;

}

void area(){

System.out.println("Area of Circle "+(3.14 \* r \* r));

}

}

class AbstractMtd{

public static void main(String [] args){

Shapee s;

s = new Rect(10,20);

s.area(); // Area of Rectangle 200;

}

}

* final keyword

1. To create constant field :

If the field is declared with final modifier, then it must be initialize at the time of declaration and we can't change the value of final field.

class math {

final static double PI = 3.14;

}

1. To prevent method overriding

There may be a situation, when we want to prevent method overriding in subclass. To disallow a method from being overridden in subclass, specify final keyword as a modifier in front of the method declaration, in super class. Methods declared as final cannot be overridden in subclass.

class A {

final void display () { /// Now this method can’t be Overrided

System.out.println ("This is final method")

}

}

1. To prevent from Inheritance

Sometimes we want to prevent a class from being inherited. To do this precedes the class declaration with keyword final. Declaring a class as final, implicitly declares all of its method as final too.

For example :

final class A

{

// Now this class can’t be Overrided

}

1. Difference Between Final And Abstract Class

|  |  |  |
| --- | --- | --- |
| Feature | Abstract Class | Final Class |
| Inheritance | Can be inherited by other classes | Cannot be inherited by other classes |
| Instantiation | Cannot be directly instantiated (objects cannot be created) | Can be directly instantiated (objects can be created) |
| Abstract Methods | Can contain abstract methods | Cannot contain abstract methods |
| Final Methods | Can contain final methods. | Cannot contain abstract methods |

* Access Specifier in Java

Java has Four types of Access Specifier

1. Private :- Private members are access only within the class.
2. Default :- If you don’t use any access specifier, it is treated as default, by default. The default members are accessible only within class and package.
3. Protected :- The protected members are accessible within package and outside the package but through inheritance only.
4. Public :- The public members are accessible everywhere, it has the widest scope among all other modifiers

class SampleP{

int x = 10; // Default Access Specifier

private name = "Pratham" // Private

void show(){ // Default

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

}}

* Object Class in Java
* The object class is the parent class of all the classes in Java by default.
* In other words, it is the topmost class of Java.
* Every class of Java is either directly or indirectly inherited from object class.
* The object class is beneficial, if you want to refer any object whose type you don’t know at run-time.
* Methods :-
  + Clone () : - Creates the clone of the object
  + Equals() :- this method checks whether two objects are equal or not.
  + Finalize() :- This method is called by the garbage collector when object is destroyed

form the memory

* getClass() :- Returns the run-time class of an object.
* toString() :- Returns a string representation of an object.
* Program

class ObjectClass {

int x = 10;

void show(){

System.out.println("X :- "+x);

}

protected void finalize(){

System.out.println("Inside Finalize Method");

}

public static void main(String [] args){

ObjectClass obj1 = new ObjectClass();

ObjectClass obj2 = new ObjectClass();

obj1.finalize();

System.out.println(obj1.equals(obj2)); // False

System.out.println(obj1.getClass()); // ObjectClass

System.out.println("ToString() :- "+obj1.toString()); // ObjectClass@70dea4e

}

}

* Interfaces in Java
* An interface is just same as abstract class.
* The main difference between an interface and abstract class is, interface is fully unimplemented, whereas abstract class is partially unimplemented.
* It means by default all the methods in an interface are abstract.
* Properties of Interface
  + By default all the methods of an interface are abstract
  + The default access specifier for an interface is public
  + Object of an interface cannot be created.
  + Reference of the interface can be created.
  + Inside the interface the field is static by default.
  + One interface can implement another interface.
  + Reference of interface can refer to the object of the class in which it is implemented but it can access only the those member that came from the interface.
* Difference Between Abstract Class and Interface

|  |  |
| --- | --- |
| Abstract Class | Interface |
| Abstract class can have abstract and non-abstract Interface. | Interface can have only abstract methods. |
| It is partially unimplemented | It is fully un-implemented. |
| It may contain instance field | It may not contain instance field |
| It contains Constructor | It may not contains Constructor |
| Default Access specifier is Default | Default Access specifier in public |
| Abstract class can be extended to subclass | Interface is always implemented by class |

* Inner Class in Java
* If the class is declared inside another class, then it is referred as inner class.
* Inner class means one class which is a member of another class.
* Properties of Inner class:
  + The object of inner class can be created only in its outer class.
  + It can access all the members of outer class.
  + Inner classes are scope to the class used to declare them.
  + Inner classes are very convenient to write event driven programs.
* Program

class SampleP{

int x = 10;

void show(){

System.out.println("Hello From Outer Show Method ");

}

class InnerClass{

InnerClass(){

System.out.println("Outer X :- "+x);

show();

}

}

InnerClass in = new InnerClass();

public static void main(String [] args){

SampleP p1 = new SampleP();

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

}

}

* Local Inner Class in Java
* If the class in declared inside the method of the outer class , then it referred as local inner class.
* Properties of Local Inner Class
  + The object of local inner class can be created in the method in which it is declared.
  + The scope of local inner class is restricted to the block.
  + A local class has access to the members of its outer class.
  + It can also access parameter and local variable of the method in which it is declared.
* Program

class LocalInnerClassP{

String name = "Prathamesh Patil";

void LocalInnerClassMtd(){

class Sample{

Sample(){

System.out.println("Hello From Sample :- Local Inner Class");

}

}

Sample s = new Sample();

}

public static void main(String [] a){

LocalInnerClassP p1 = new LocalInnerClassP();

p1.LocalInnerClassMtd();

}

}

* Anonymous Class in Java
* The class which does not have any name is referred as anonymous class.
* Properties of Anonymous class
  + It is an Inner class without a name.
  + Only a single object can be created of anonymous class.
  + Constructors are not allowed in anonymous class
  + It always created by either implementing interface or by extending a super class.
* Anonymous Inner classes are useful in Windows programming

interface Sample{

void show();

void display();

}

class AnonymousP{

public static void main(String [] args){

Sample s = new Sample(){

public void show(){

System.out.println("Inside Show Method");

}

public void display(){

System.out.println("Inside Display Method");

}

};

s.show();

s.display();

}}

* Wrapper Class in Java
* A wrapper class is a class whose object wraps or contains primitive data type.
* In other words we can wrap a primitive data type into a wrapper class object.
* All the wrapper classes are declared in java-lang package.
* Data structures in the collection framework of Java such as Vector or ArrayList stores only objects and not primitive types. So to convert primitive types into objects or vice versa Wrapper classes are used.

|  |  |
| --- | --- |
| Primitive Type | Wrapper Class |
| boolean | Boolean |
| byte | Byte |
| short | Short |
| char | Character |
| int | Integer |
| long | Long |
| float | Float |
| double | Double |

* **Autoboxing** is the automatic conversion that the Java compiler makes between the primitive types and their corresponding object wrapper classes. For example, converting an **int** to an **Integer**.
* **Unboxing** is the reverse process where the wrapper class object is converted back to a primitive type. For example, converting an **Integer** to an **int**.
* Methods Of Wrapper Classes

1. Converting Primitive Type to Wrapper Object ( **Integer intObj = new Integer (i);** )
2. Converting Wrapper Object to Primitive Type **( int i = intObj.int+-Value (); )**
3. Converting Primitive type to String Type **( String str = Integer.toString(i); )**
4. Converting String type to Primitive Type **( int i = Integer.parseInt(str); )**
5. Converting string to wrapper objecting  **( Integer intObj = Integer.valueOf (str); )**

* Exception Handling
* An error may produce an incorrect output or may terminate the execution of the program or even may cause the application to crash.
* It is therefore important to detect and manage properly all the possible error conditions in the program so that the program will not terminate or crash during execution.

1. Compile Time Errors:-

* All syntax errors will be detected and displayed by the Java compiler and therefore these errors are known as compile time errors.
* Whenever the compiler displays an error, it will not create the .class file. It is therefore necessary that we fix all the errors before we can successfully compile and run the program. Most of the compile time errors are due to typing mistakes.
* The most common problems are

a. Missing semicolons

b. Missing (or mismatch) brackets in classes and methods.

c. Misspelling of identifiers and keywords.

d. Missing double quotes in string.

e. Use of undeclared variable.

f. Bad reference to object. g. And many more

1. Run Time Erros:-

* Sometimes, a program may compile successfully creating the .class file but may not run properly. Such programs may produce wrong results due to wrong logic or may terminate in between. Error which occurs at run time is called as Exception.

1. ArithmeticException It occurs due to dividing number by zero.
2. ArrayIndexOutOfBoundsException It occurs when we access an array element which is out of the bounds of an array
3. NegativeArraySizeException It occurs when we use to create an array of negative size.
4. NullPointerException It occurs by referencing null object.

public static void main(String [] args) {

String str=null; //referencing null object

System.out.println("Length of string is :"+str.length()); // Nullptr Exception

}

1. NumberFormatException It occurs when conversion between string and number fails.

public static void main(String [] args) {

String str="ABCD";

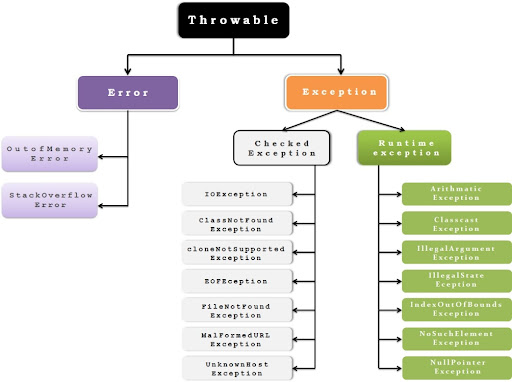
int x = Integer.parseInt(str);

System.out.println("x="+x);

}

1. StringIndexOutOfBoundsException :- It occurs when a program attempts to access a nonexistent character position in a string.
2. IOException It occurs due to general input and output failure.
3. FileNotFoundException It occurs due to opening a non existing file.

* Exception Hierarchy And Custom Exception
* All exception and errors are the subclasses of class Throwable class, which is base class in exception Hierarchy.
* One branch is headed by Exception this class is used for Exceptional condition, that user program should catch.
* Another branch Error is used by the Java run-time system (JVM) to indicate error to generate within JVM. We can't handle errors.



* Custom Exceptions

class InvalidAgeException extends Exception{

InvalidAgeException(String str){

super(str);

}

}

class CustomExceptions{

public static void main(String [] args){

try{

System.out.println("Enter your Age");

Scanner in = new Scanner(System.in);

int age = in.nextInt();

if(age < 18){

throw new InvalidAgeException("You Cannot Drive Vehicle");

}else{

System.out.println("You can Drive Vehicle");

}

}catch(Exception e){

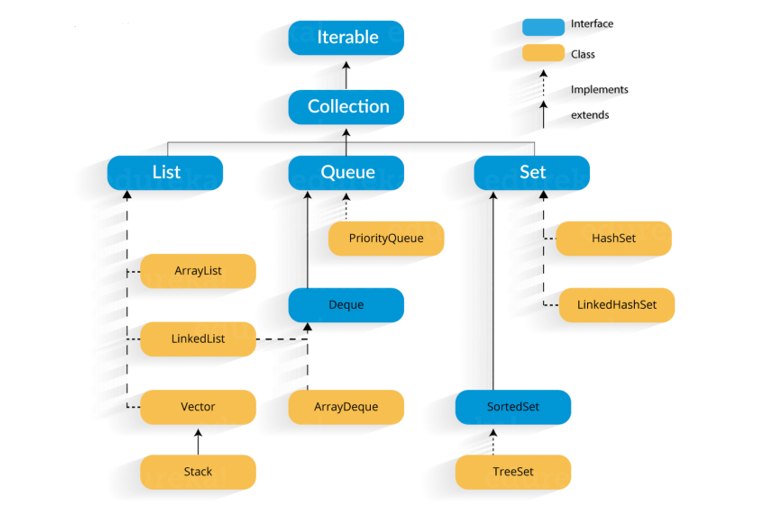
System.out.println(e);

}

}

}

* Collection Framework



Difference Between C++ and Java

|  |  |
| --- | --- |
| C++ | Java |
| It is platform dependent | It is platform independent |
| It used ASCII | It used Unicode |
| It dose not have built In Multithreading | It have built in support for Multithreading |
| It is Compiled | It is Compiled and Interpreted |
| It supports Operator Overloading | It does not Support operator overloading |
| It support pointers | It does not support pointers |
| It does not support manual memory allocation or deallocation | It support manual memory allocation and de-allocation |