**OOPS:**

Object-Oriented Programming is a methodology or paradigm to design a program using classes and objects. It simplifies the software development and maintenance by providing some concepts:

* Object
* Class
* Inheritance
* Polymorphism
* Abstraction
* Encapsulation

**Object**

Any entity that has state and behavior is known as an object. For example: chair, pen, table, keyboard, bike etc. It can be physical and logical.

**Class**

Collection of objects is called class. It is a logical entity.

A class is a group of objects that has common properties. It is a template or blueprint from which objects are created.

**Inheritance**

When one object acquires all the properties and behaviours of parent object i.e. known as inheritance. It provides code reusability. It is used to achieve runtime polymorphism.

**Polymorphism**

When one task is performed by different ways i.e. known as polymorphism. For example: to convense the customer differently, to draw something e.g. shape or rectangle etc.

In java, we use method overloading and method overriding to achieve polymorphism.

Another example can be to speak something e.g. cat speaks meaw, dog barks woof etc.

**Abstraction**

Hiding internal details and showing functionality is known as abstraction. For example: phone call, we don't know the internal processing.

In java, we use abstract class and interface to achieve abstraction.

**Encapsulation**

Binding (or wrapping) code and data together into a single unit is known as encapsulation. For example: capsule, it is wrapped with different medicines.

A java class is the example of encapsulation. Java bean is the fully encapsulated class because all the data members are private here.

**Types of Variable**

|  |  |
| --- | --- |
| Local Variable: A variable that is declared inside the method is called local variable. | |
| Instance Variable: A variable that is declared inside the class but outside the method is called instance variable . It is not declared as static. | | |
| Static variable: A variable that is declared as static is called static variable. It cannot be local. |

**Data Types in Java**

|  |
| --- |
| In java, there are two types of data types  primitive data types  non-primitive data types |



**Java static keyword:**

The static keyword in java is used for memory management mainly. We can apply java static keyword with variables, methods, blocks and nested class. The static keyword belongs to the class than instance of the class.

The static can be:

variable (also known as class variable)

method (also known as class method)

block

nested class

1)The static method can not use non static data member or call non-static method directly.

2)this and super cannot be used in static context.

**String :**

String s1="Welcome";

String s=new String("Welcome")

String objects are stored in a special memory area known as string constant pool.

In java, string objects are immutable. Immutable simply means unmodifiable or unchangeable

**StringBuffer:**

StringBuffer class is used to created mutable (modifiable) string

Java StringBuffer class is thread-safe i.e. multiple threads cannot access it simultaneously. So it is safe and will result in an order.

|  |  |  |
| --- | --- | --- |
| No. | **String** | **StringBuffer** |
| 1) | String class is immutable. | StringBuffer class is mutable. |
| 2) | String is slow and consumes more memory when you concat too many strings because every time it creates new instance. | StringBuffer is fast and consumes less memory when you cancat strings. |
| 3) | String class overrides the equals() method of Object class. So you can compare the contents of two strings by equals() method. | StringBuffer class doesn't override the equals() method of Object class. |

|  |  |  |
| --- | --- | --- |
| No. | **StringBuffer** | **StringBuilder** |
| 1) | StringBuffer is synchronized i.e. thread safe. It means two threads can't call the methods of StringBuffer simultaneously. | StringBuilder is non-synchronized i.e. not thread safe. It means two threads can call the methods of StringBuilder simultaneously. |
| 2) | StringBuffer is less efficient than StringBuilder. | StringBuilder is more efficient than StringBuffer. |

|  |
| --- |
| **Methods with Description** |
| char **charAt**(int index) Returns the character at the specified index. |
| int **compareTo**(Object o) Compares this String to another Object. |
| int **compareTo**(String anotherString) Compares two strings lexicographically. |
| int **compareToIgnoreCase**(String str) Compares two strings lexicographically, ignoring case differences. |
| String **concat**(String str) Concatenates the specified string to the end of this string. |
| boolean **contentEquals**(StringBuffer sb)  Returns true if and only if this String represents the same sequence of characters as the specified StringBuffer. |
| static String **copyValueOf**(char[] data) Returns a String that represents the character sequence in the array specified. |
| static String **copyValueOf**(char[] data, int offset, int count) Returns a String that represents the character sequence in the array specified. |
| boolean **endsWith**(String suffix) Tests if this string ends with the specified suffix. |
| boolean **equals**(Object anObject) Compares this string to the specified object. |
| boolean **equalsIgnoreCase**(String anotherString) Compares this String to another String, ignoring case considerations. |
| byte **getBytes**() Encodes this String into a sequence of bytes using the platform's default charset, storing the result into a new byte array. |
| byte[] **getBytes**(String charsetName)  Encodes this String into a sequence of bytes using the named charset, storing the result into a new byte array. |
| void **getChars**(int srcBegin, int srcEnd, char[] dst, int dstBegin) Copies characters from this string into the destination character array. |
| int **hashCode**() Returns a hash code for this string. |
| int **indexOf**(int ch) Returns the index within this string of the first occurrence of the specified character. |
| int **indexOf**(int ch, int fromIndex) Returns the index within this string of the first occurrence of the specified character, starting the search at the specified index. |
| int **indexOf**(String str) Returns the index within this string of the first occurrence of the specified substring. |
|  |
| int **indexOf**(String str, int fromIndex)  Returns the index within this string of the first occurrence of the specified substring, starting at the specified index |
| String **intern**() Returns a canonical representation for the string object. |
| int **lastIndexOf**(int ch) Returns the index within this string of the last occurrence of the specified character. |
| int **lastIndexOf**(int ch, int fromIndex)  Returns the index within this string of the last occurrence of the specified character, searching backward starting at the specified index. |
| int **lastIndexOf**(String str) Returns the index within this string of the rightmost occurrence of the specified substring. |
| int **lastIndexOf**(String str, int fromIndex)  Returns the index within this string of the last occurrence of the specified substring, searching backward starting at the specified index. |
| int **length**() Returns the length of this string. |
| boolean matches(String regex) Tells whether or not this string matches the given regular expression. |
| String **replace**(char oldChar, char newChar)  Returns a new string resulting from replacing all occurrences of oldChar in this string with newChar. |
| String **replaceAll**(String regex, String replacement)  Replaces each substring of this string that matches the given regular expression with the given replacement. |
| String **replaceFirst**(String regex, String replacement)  Replaces the first substring of this string that matches the given regular expression with the given replacement. |
| String[] **split**(String regex) Splits this string around matches of the given regular expression. |
| String[] **split**(String regex, int limit) Splits this string around matches of the given regular expression. |
| boolean **startsWith**(String prefix) Tests if this string starts with the specified prefix. |
| boolean **startsWith**(String prefix, int toffset) Tests if this string starts with the specified prefix beginning a specified index. |
| CharSequence **subSequence**(int beginIndex, int endIndex) Returns a new character sequence that is a subsequence of this sequence. |
| String **substring**(int beginIndex) Returns a new string that is a substring of this string. |
| String **substring**(int beginIndex, int endIndex) Returns a new string that is a substring of this string. |
| char[] **toCharArray**() Converts this string to a new character array. |
| String **toLowerCase**() Converts all of the characters in this String to lower case using the rules of the default locale. |
| String **toString**() This object (which is already a string!) is itself returned. |
| String **toUpperCase**() Converts all of the characters in this String to upper case using the rules of the default locale. |
| String **trim**() Returns a copy of the string, with leading and trailing whitespace omitted. |
| static String **valueOf**(primitive data type x) Returns the string representation of the passed data type argument. |

**Exceptions:**

1) Checked Exception

The classes that extend Throwable class except RuntimeException and Error are known as checked exceptions e.g.IOException, SQLException etc. Checked exceptions are checked at compile-time.

2) Unchecked Exception

The classes that extend RuntimeException are known as unchecked exceptions e.g. ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException etc. Unchecked exceptions are not checked at compile-time rather they are checked at runtime.

3) Error

Error is irrecoverable e.g. OutOfMemoryError, VirtualMachineError, AssertionError etc.

There are 5 keywords used in java exception handling.

try

catch

finally

throw

throws

For each try block there can be zero or more catch blocks, but only one finally block.

EX:

public class TestFinallyBlock2{

  public static void main(String args[]){

  try{

   int data=25/0;

   System.out.println(data);

  }

  catch(ArithmeticException e){System.out.println(e);}

  finally{System.out.println("finally block is always executed");}

  System.out.println("rest of the code...");

  }

}

EX:

void m()throws ArithmeticException{

throw new ArithmeticException("sorry");

}

|  |  |  |
| --- | --- | --- |
| No. | **throw** | **throws** |
| 1) | Java throw keyword is used to explicitly throw an exception. | Java throws keyword is used to declare an exception. |
| 2) | Checked exception cannot be propagated using throw only. | Checked exception can be propagated with throws. |
| 3) | Throw is followed by an instance. | Throws is followed by class. |
| 4) | Throw is used within the method. | Throws is used with the method signature. |
| 5) | You cannot throw multiple exceptions. | You can declare multiple exceptions e.g. public void method()throws IOException,SQLException. |

|  |  |  |  |
| --- | --- | --- | --- |
| No. | **final** | **finally** | **finalize** |
| 1) | Final is used to apply restrictions on class, method and variable. Final class can't be inherited, final method can't be overridden and final variable value can't be changed. | Finally is used to place important code, it will be executed whether exception is handled or not. | Finalize is used to perform clean up processing just before object is garbage collected. |
| 2) | Final is a keyword. | Finally is a block. | Finalize is a method. |

If the superclass method does not declare an exception

If the superclass method does not declare an exception, subclass overridden method cannot declare the checked exception but it can declare unchecked exception.

If the superclass method declares an exception

If the superclass method declares an exception, subclass overridden method can declare same, subclass exception or no exception but cannot declare parent exception.

**Multi Threading:**

**Life Cycle:**

1. New
2. Runnable
3. Running
4. Non-Runnable (Blocked)
5. Terminated

There are two ways to create a thread:

1. By extending Thread class
2. By implementing Runnable interface.

**Methods:**

public void **run**(): is used to perform action for a thread.

public void **start**(): starts the execution of the thread.JVM calls the run() method on the thread.

public void **sleep**(long miliseconds): Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds.

public void **join**(): waits for a thread to die.

public void **join**(long miliseconds): waits for a thread to die for the specified miliseconds.

public int **getPriority**(): returns the priority of the thread.

public int **setPriority**(int priority): changes the priority of the thread.

public String **getName**(): returns the name of the thread.

public void **setName**(String name): changes the name of the thread.

public Thread **currentThread**(): returns the reference of currently executing thread.

public int **getId**(): returns the id of the thread.

public Thread.State **getState**(): returns the state of the thread.

public boolean **isAlive**(): tests if the thread is alive.

public void **yield**(): causes the currently executing thread object to temporarily pause and allow other threads to execute.

public void **suspend**(): is used to suspend the thread(depricated).

public void **resume**(): is used to resume the suspended thread(depricated).

public void **stop**(): is used to stop the thread(depricated).

public boolean **isDaemon**(): tests if the thread is a daemon thread.

public void **setDaemon**(boolean b): marks the thread as daemon or user thread.

public void **interrupt**(): interrupts the thread.

public boolean **isInterrupted**(): tests if the thread has been interrupted.

public static boolean **interrupted**(): tests if the current thread has been interrupted.

class Multi extends Thread{

public void run(){

System.out.println("thread is running...");

}

public static void main(String args[]){

Multi t1=new Multi();

t1.start();

 }

}

**What is the base class of all classes?**

java.lang.Object

**Why char uses 2 byte in java and what is \u0000 ?**

java uses unicode system rather than ASCII code system. \u0000 is the lowest range of unicode system.

|  |
| --- |
| lowest value:\u0000 |
| highest value:\uFFFF |
|  |

**What are the different ways to create an object in Java?**

|  |
| --- |
| There are many ways to create an object in java. They are:  By new keyword  By newInstance() method  By clone() method  By factory method etc.  object gets the memory in Heap area  reference variables the memory in stack  **Method Overloading:** |

* In java, Methood Overloading is not possible by changing the return type of the method.
* One type is promoted to another implicitly if no matching datatype is found
* One type is not de-promoted implicitly for example double cannot be depromoted to any type implicitely.
* If there is no constructor in a class, compiler automatically creates a default constructor.
* Constructor must not have return type.
* Call to this() must be the first statement in constructor.

**Inheritance:**

* extends
* Multiple inheritance is not supported in java through class.
* super is used to refer immediate parent class instance variable.
* super() is used to invoke immediate parent class constructor.
* super is used to invoke immediate parent class method.
* super() is added in each class constructor automatically by compiler.

**Can we override static method?**

No, static method cannot be overridden

**Can we override java main method?**

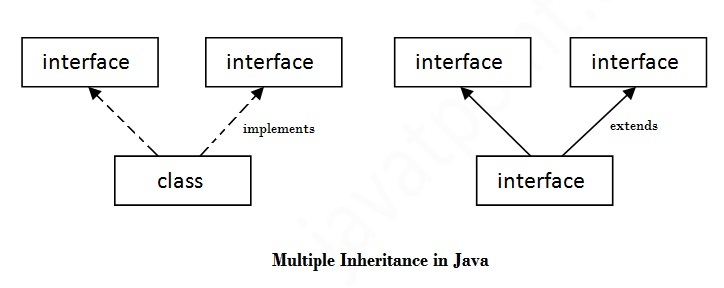
No, because main is a static method.

**Abstract class:**

* If there is any abstract method in a class, that class must be abstract.
* If you are extending any abstract class that have abstract method, you must either provide the implementation of the method or make this class abstract.
* An abstract class can have data member, abstract method, method body, constructor and even main() method.

**Interface:**

The java compiler adds public and abstract keywords before the interface method and public, static and final keywords before data members.



**Difference between abstract class and interface:**

|  |  |
| --- | --- |
| **Abstract class** | **Interface** |
| 1) Abstract class can have abstract and non-abstract methods. | Interface can have only abstract methods. |
| 2) Abstract class doesn't support multiple inheritance. | Interface supports multiple inheritance. |
| 3) Abstract class can have final, non-final, static and non-static variables. | Interface has only static and final variables. |
| 4) Abstract class can have static methods, main method and constructor. | Interface can't have static methods, main method or constructor. |
| 5) Abstract class can provide the implementation of interface. | Interface can't provide the implementation of abstract class. |
| 6) The abstract keyword is used to declare abstract class. | The interface keyword is used to declare interface. |
| 7) Example: public abstract class Shape{ public abstract void draw(); } | Example: public interface Drawable{ void draw(); } |

**Access Modifiers:**

private

default

protected

public

* A class cannot be private or protected except nested class.
* The default modifier is accessible only within package.
* The protected access modifier is accessible within package and outside the package but through inheritance only.
* The protected access modifier can be applied on the data member, method and constructor. It can't be applied on the class.

**Wrapper Class:**

Boolean, Character, Byte, Short, Integer, Long, Float, Double

**Autoboxing**  :primitive into object

**Unboxing** : object into primitive

**Character Occurrence:**

public static void characterCount(String inputString){

      Map<Character, Integer> map=new LinkedHashMap<Character, Integer>();

      char[] inputAr=inputString.toCharArray();

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

          char ch=inputAr[i];

          if(map.containsKey(ch))

                 map.put(ch, map.get(ch) +1);

          else

                 map.put(ch, 1);

      }

      Iterator<Character> charIterator=map.keySet().iterator();

      while(charIterator.hasNext()){

          char ch=charIterator.next();

          System.out.print(ch+"="+map.get(ch)+" ");

      }

   }