* Inheretance – is a relationship – parent child relationship.
* Multiple inheritance is not possible that means a extending both b and c because they might have same functions and java will get confused which to follow so ambiguity.
* Aggregation – has a relationship – which means one object can have a parameter which has another object as one of the parameter of that object.
* Overloading – having same methods with different parameters but return type should be same because different return type is not allowed in method overloading it will give compile time error.main method can be overloaded but it will accept only that method which has sting array arguments as input.
* Overriding – means one method which has many implementation. Static method cannot be overridden because it is associated with a class but not with the object.
* Static methods are accessible via class name.
* Method overriding can have different return types this is useful for checking type of the return because child class can change the return type to its own class so that if child class executes function which returns this we will understand that the return type is actually which class it will become problem if there are multiple level inherentaance if return type is same.
* Final keyword can be assigned to variable clas and method. Final variable cannot be changed it is constant blank final variable can be assigned value only inconstructor. Final method cannot be oveerriden. Blank static final variable will be assigned value only in static block. Static class cannot be xtended.
* Static and final both methods cannot be overridden.

Points to Remember for abstract class

* An abstract class must be declared with an abstract keyword.
* It can have abstract and non-abstract methods.
* It cannot be instantiated.
* It can have [constructors](https://www.javatpoint.com/java-constructor)

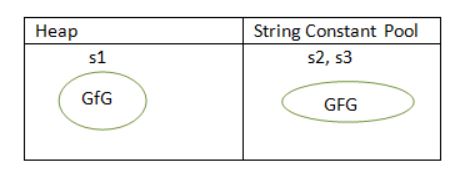
and static methods also.

* It can have final methods which will force the subclass not to change the body of the method.
* String is immutable that means it cannot be changed. For mutable strings we use stringBuffer and stringbuilder.
* Two ways string can be created – by literal and by new keyword.
* When created by literal if same string is created it will check in string constant pool and will not create a new object it will refer to the previous one. And by new keyword it will create new ref but it will refer to same thing in string constant pool.
* Strings can be compared using equals, equalIgnoreCase, CompareTo methods.
* Strings can be concatenated also, it can be concatenated using concat, join StringBuilder.append and Strongjoiner, Collectors.joinning methods.
* Substring can be provided in java by using Substring(firstIndex, lastIndex) where firstIndex will be included and last index will be excluded.
* Split can be used to split strings with provided regex and limit.

**intern() method :**In Java, when we perform any operation using intern() method, it returns a canonical representation for the string object. A pool is managed by String class.

* When the intern() method is executed then it checks whether the String equals to this String Object is in the pool or not.
* If it is available, then the string from the pool is returned. Otherwise, this String object is added to the pool and a reference to this String object is returned.
* It follows that for any two strings s and t, s.intern() == t.intern() is true if and only if s.equals(t) is true.

|  |
| --- |
| // Java program to illustrate  // intern() method  class GFG {      public static void main(String[] args)      {          // S1 refers to Object in the Heap Area          String s1 = new String("GFG"); // Line-1            // S2 refers to Object in SCP Area          String s2 = s1.intern(); // Line-2            // Comparing memory locations          // s1 is in Heap          // s2 is in SCP          System.out.println(s1 == s2);            // Comparing only values          System.out.println(s1.equals(s2));            // S3 refers to Object in the SCP Area          String s3 = "GFG"; // Line-3            System.out.println(s2 == s3);      }  } |

* Output:
* false
* true
* true
* **Explanation :** Whenever we create a String Object, two objects will be created i.e. One in the Heap Area and One in the String constant pool and the String object reference always points to heap area object. When line-1 execute, it will create two objects and pointing to the heap area created object. Now when line-2 executes, it will refer to the object which is in the SCP. Again when line-3 executes, it refers to the same object which is in the SCP area because the content is already available in the SCP area. No need to create a new one object.
* [](https://media.geeksforgeeks.org/wp-content/cdn-uploads/String-interning-in-Java-1.jpg)
* If the corresponding String constant pool(SCP) object is not available then intern() method itself will create the corresponding SCP object.

|  |  |  |
| --- | --- | --- |
| **No.** | **String** | **StringBuffer** |
| 1) | The String class is immutable. | The StringBuffer class is mutable. |
| 2) | String is slow and consumes more memory when we concatenate too many strings because every time it creates new instance. | StringBuffer is fast and consumes less memory when we concatenate t 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. |
| 4) | String class is slower while performing concatenation operation. | StringBuffer class is faster while performing concatenation operation. |
| 5) | String class uses String constant pool. | StringBuffer uses Heap memory |

* Synchronous – thread safe – one thread at a time.
* Non synchronous – not thread safe – multiple threads share at a same time.
* Static anf final cane be overloaded but cannot be overridden.
* Java Regex – Its api that is used for string validation or manipulation.

Pattern.matches(regex, char seq);

e.g. – [0-9] includes 0 to 9 and [^0-9] without 0-9 {} indicates number of characters

|  |  |
| --- | --- |
| . | Any character (may or may not match terminator) |
| \d | Any digits, short of [0-9] |
| \D | Any non-digit, short for [^0-9] |
| \s | Any whitespace character, short for [\t\n\x0B\f\r] |
| \S | Any non-whitespace character, short for [^\s] |
| \w | Any word character, short for [a-zA-Z\_0-9] |
| \W | Any non-word character, short for [^\w] |
| \b | A word boundary |
| \B | A non word boundary |

* Exception –



* Difference between throw and throws –

|  |  |  |  |
| --- | --- | --- | --- |
| **Sr. no.** | **Basis of Differences** | **throw** | **throws** |
| 1. | Definition | Java throw keyword is used throw an exception explicitly in the code, inside the function or the block of code. | Java throws keyword is used in the method signature to declare an exception which might be thrown by the function while the execution of the code. |
| 2. | Type of exception Using throw keyword, we can only propagate unchecked exception i.e., the checked exception cannot be propagated using throw only. | Using throws keyword, we can declare both checked and unchecked exceptions. However, the throws keyword can be used to propagate checked exceptions only. |  |
| 3. | Syntax | The throw keyword is followed by an instance of Exception to be thrown. | The throws keyword is followed by class names of Exceptions to be thrown. |
| 4. | Declaration | throw is used within the method. | throws is used with the method signature. |
| 5. | Internal implementation | We are allowed to throw only one exception at a time i.e. we cannot throw multiple exceptions. | We can declare multiple exceptions using throws keyword that can be thrown by the method. For example, main() throws IOException, SQLException. |

* Throws keyword can be used to declare checked and unchecked both the exceptions but it will propagate only checked exceptions.
* Throw keywprd can be used to declare only unchecked exceptions.

Some of the rules are listed below:

* **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.

|  |  |
| --- | --- |
| **Type** | **Description** |
| [Member Inner Class](https://www.javatpoint.com/member-inner-class) | A class created within class and outside method. |
| [Anonymous Inner Class](https://www.javatpoint.com/anonymous-inner-class) | A class created for implementing an interface or extending class. The java compiler decides its name. |
| [Local Inner Class](https://www.javatpoint.com/local-inner-class) | A class was created within the method. |
| [Static Nested Class](https://www.javatpoint.com/static-nested-class) | A static class was created within the class. |
| [Nested Interface](https://www.javatpoint.com/nested-interface) | An interface created within class or interface. |

There are given some points that should be remembered by the java programmer.

* The nested interface must be public if it is declared inside the interface, but it can have any access modifier if declared within the class.
* Nested interfaces are declared static
* A thread is basically a saperate execution. It is a light weight subprocess. A process can have multiple threads. A process has dedicated address in memory space whereas threads share the memory but each thread requires at least one process to execute.
* Threads can be created using two methods either by extending Thread class or implementing Runnable class. Both can have run function in it. So whenever we do start() it will run run() method and perform action written in it. Also whenever you create thread using runnable interface you have to create thread like this,

Thread t1 = new Thread(Runnable class object). So this will actually create a thread. And class objects which are extending Thread are by default treated as thread.

* Thread scheduling is done to schedule threads which are in the runnable state and which should be picked first. Now there are 3 diff strategies to decide that,
  1. First come first serve strategy – where the thread which came first will be served but in that case also it can lead to starvation(like one thread occupying a space for a long time and blocking critical part of the code for long) so we use time slicing.
  2. Time slicing – here every thread has assigned a particular time. And the thread has to leave a CPU after that time is completed.
  3. Priority based – whichever thread has higher priority between 1 to 10 that get executed first.
* Whenever a particular thread goes in the sleep mode thread scheduler will immediately pick the other thread in the queue and starts executing.
* **class** TestSleepMethod1 **extends** Thread{
* **public** **void** run(){
* **for**(**int** i=1;i<5;i++){
* // the thread will sleep for the 500 milli seconds
* **try**{Thread.sleep(500);}**catch**(InterruptedException e){System.out.println(e);}
* System.out.println(i);
* }
* }
* **public** **static** **void** main(String args[]){
* TestSleepMethod1 t1=**new** TestSleepMethod1();
* TestSleepMethod1 t2=**new** TestSleepMethod1();
* t1.start();
* t2.start();
* }
* }

**Output:**

1

1

2

2

3

3

4

4

# **What if we call Java run() method directly instead start() method?**

* Each thread starts in a separate call stack.
* Invoking the run() method from the main thread, the run() method goes onto the current call stack rather than at the beginning of a new call stack.
* So instead of creating a saperate stack for thread, it will go into the main stack. And lets say there are 2 threads and run method has for loo[s and it is sleeping thread for 500ms so forst thread will be executed completely with 500ms gaps and then second threads will be executed because it is treated as objects in main stack and not like thread so no context switching.
* By default priority given to thread will be 5. And we can assign a different priority and then can get the priority Thread().currentThread.getPriority().
* Deamon thread is a service provider thread that provides background services to user thead. So as soon as all user threads are dead JVM will automatically terminate the thread. .setDeamon is used to set the thread as deamon and isDeamon is used to check whether it’s a deamo thread or not.
* The CPU always runs only one thread at a time but the context switching between the thread is so fast that we feel that there are two threads running simultaneously.
* Examples –

1. A very good example of thread-based multithreading is a word processing program that checks the spelling of words in a document while writing the document. This is possible only if each action is performed by a separate thread.

2. Another familiar example is a browser that starts rendering a web page while it is still downloading the rest of page.

* Gafrbage collection is used in java to automatically destroy unreferenced objects, this is done in java automatically that’s why java is more memory effeicient.

By unreferenced objects means objects which are initialised as null, objects which are assigned ref to another object so the previous ref is destroyed and anonymous objects(new Class).

Finalize() method is used before destroying the object. We can call gc() to invoke garbage collector in class and we have to implement finalize() function to do any cleanup that we need to do before destroying object.

Garbage collector destroys objects only those which are created by new keyword. For others we will have to invoke finalize () method explicitly.

1. // Java program that prints the odd and even numbers using two threads.
2. // the time complexity of the program is O(N), where N is the number up to which we
3. // are displaying the numbers
4. **public** **class** OddEvenExample
5. {
6. // Starting the counter
7. **int** contr = 1;
8. **static** **int** NUM;
9. // Method for printing the odd numbers
10. **public** **void** displayOddNumber()
11. {
12. // note that synchronized blocks are necessary for the code for getting the desired
13. // output. If we remove the synchronized blocks, we will get an exception.
14. **synchronized** (**this**)
15. {
16. // Printing the numbers till NUM
17. **while** (contr < NUM)
18. {
19. // If the contr is even then display
20. **while** (contr % 2 == 0)
21. {
22. // handling the exception handle
23. **try**
24. {
25. wait();
26. }
27. **catch** (InterruptedException ex)
28. {
29. ex.printStackTrace();
30. }
31. }
32. // Printing the number
33. System.out.print(contr + " ");
34. // Incrementing the contr
35. contr = contr + 1;
36. // notifying the thread which is waiting for this lock
37. notify();
38. }
39. }
40. }
41. // Method for printing the even numbers
42. **public** **void** displayEvenNumber()
43. {
44. **synchronized** (**this**)
45. {
46. // Printing the number till NUM
47. **while** (contr < NUM)
48. {
49. // If the count is odd then display
50. **while** (contr % 2 == 1)
51. {
52. // handling the exception
53. **try**
54. {
55. wait();
56. }
57. **catch** (InterruptedException ex)
58. {
59. ex.printStackTrace();
60. }
61. }
62. // Printing the number
63. System.out.print(contr + " ");
64. // Incrementing the contr
65. contr = contr +1;
66. // Notifying to the 2nd thread
67. notify();
68. }
69. }
70. }
71. // main method
72. **public** **static** **void** main(String[] argvs)
73. {
74. // The NUM is given
75. NUM = 20;
76. // creating an object of the class OddEvenExample
77. OddEvenExample oe = **new** OddEvenExample();
78. // creating a thread th1
79. Thread th1 = **new** Thread(**new** Runnable()
80. {
81. **public** **void** run()
82. {
83. // invoking the method displayEvenNumber() using the thread th1
84. oe.displayEvenNumber();
85. }
86. });
87. // creating a thread th2
88. Thread th2 = **new** Thread(**new** Runnable()
89. {
90. **public** **void** run()
91. {
92. // invoking the method displayOddNumber() using the thread th2
93. oe.displayOddNumber();
94. }
95. });
96. // starting both of the threads
97. th1.start();
98. th2.start();
99. }
100. }
     * Synchronised block is used so that one thread can be run at a time that means if one thread is at sleep it should wait for the sleep time and then run next task complete it and then other thread will work.

* //example of java synchronized method
* **class** Table{
* **synchronized** **void** printTable(**int** n){//synchronized method
* **for**(**int** i=1;i<=5;i++){
* System.out.println(n\*i);
* **try**{
* Thread.sleep(400);
* }**catch**(Exception e){System.out.println(e);}
* }
* }
* }
* **class** MyThread1 **extends** Thread{
* Table t;
* MyThread1(Table t){
* **this**.t=t;
* }
* **public** **void** run(){
* t.printTable(5);
* }
* }
* **class** MyThread2 **extends** Thread{
* Table t;
* MyThread2(Table t){
* **this**.t=t;
* }
* **public** **void** run(){
* t.printTable(100);
* }
* }
* **public** **class** TestSynchronization2{
* **public** **static** **void** main(String args[]){
* Table obj = **new** Table();//only one object
* MyThread1 t1=**new** MyThread1(obj);
* MyThread2 t2=**new** MyThread2(obj);
* t1.start();
* t2.start();
* }
* }

**Output:**

5

10

15

20

25

100

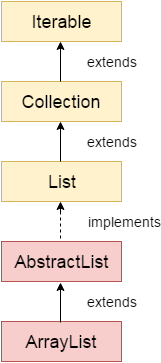
200

300

400

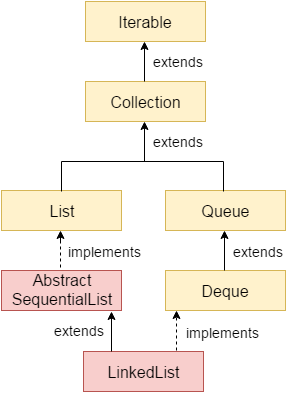
500

* + Collection is a group of objects. Collection implements iterable interface and it has many interfaces (List, Set, Queue) and classes that implements this interfaces(ArrayList, LinkedList, HashSet, LinkedHashset, vector, dequeu),
* 
  + Array List – array List is used to dynamically store value and maintain its order.it is non synchronised,



* Java ArrayList class can contain duplicate elements.
* Java ArrayList class maintains insertion order.
* Java ArrayList class is non [synchronized](https://www.javatpoint.com/synchronization-in-java).
* Java ArrayList allows random access because the array works on an index basis.
* In ArrayList, manipulation is a little bit slower than the LinkedList in Java because a lot of shifting needs to occur if any element is removed from the array list.
* We can not create an array list of the primitive types, such as int, float, char, etc. It is required to use the required wrapper class in such cases. For example:

1. ArrayList<**int**> al = ArrayList<**int**>(); // does not work
2. ArrayList<Integer> al = **new** ArrayList<Integer>(); // works fine

* Java ArrayList gets initialized by the size. The size is dynamic in the array list, which varies according to the elements getting added or removed from the list.
* Java.util package has a Collection class which has static method sort() so by using Collection.sort() we can sort the elements of arraylist.
* By using equals() method we can compare two arraylist.
* By using reverse() method we can reverse any collection.
* We can use Collections.synchronizedList(List<T>) method to synchronize collections in java. The synchronizedList(List<T>) method is used to return a synchronized (thread-safe) list backed by the specified list.
* By using Collections.reverseOrder(Comparator<T>cmp) method, we can sort the collection in reverse order.
* LinkedList –
* 

|  |  |
| --- | --- |
| **ArrayList** | **LinkedList** |
| 1) ArrayList internally uses a **dynamic array** to store the elements. | LinkedList internally uses a **doubly linked list** to store the elements. |
| 2) Manipulation with ArrayList is **slow** because it internally uses an array. If any element is removed from the array, all the bits are shifted in memory. | Manipulation with LinkedList is **faster** than ArrayList because it uses a doubly linked list, so no bit shifting is required in memory. |
| 3) An ArrayList class can **act as a list** only because it implements List only. | LinkedList class can **act as a list and queue** both because it implements List and Deque interfaces. |
| 4) ArrayList is **better for storing and accessing** data. | LinkedList is **better for manipulating** data. |