**Knockout the Interview**

**Jeremiah Haile**

**Dec 28, 2018**

|  |
| --- |
| **Useful Java Practice Links:**  <https://www.tutorialspoint.com/javaexamples/java_collections.htm>  <https://www.javatpoint.com/java-hashtable>  <https://beginnersbook.com/java-collections-tutorials/>  <https://www.edureka.co/blog/interview-questions/java-interview-questions/>  <https://jrebel.com/rebellabs/java-regular-expressions-cheat-sheet/> |



**Basic Java Questions:**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**Boxing & Unboxing**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

|  |
| --- |
| Wrapper Classes in Java  A Wrapper class is a class whose object wraps or contains a primitive data types. When we create an object to a wrapper class, it contains a field and in this field, we can store a primitive data types. In other words, we can wrap a primitive value into a wrapper class object.  Need of Wrapper Classes   1. They convert primitive data types into objects. Objects are needed if we wish to modify the arguments passed into a method (because primitive types are passed by value). 2. The classes in java.util package handles only objects and hence wrapper classes help in this case also. 3. Data structures in the Collection framework, such as [ArrayList](https://www.geeksforgeeks.org/arraylist-in-java/) and [Vector](https://www.geeksforgeeks.org/vector-vs-arraylist-java/), store only objects (reference types) and not primitive types. 4. An object is needed to support synchronization in multithreading.   Primitive Data types and their Corresponding Wrapper class    public class test1 {    public static void main (String args[]){  int i = 10;    // We need boxing and unboxing to convert primitive data to object  Integer iRef = new Integer(i); // Boxed (Constracted the objct)  int j = iRef.intValue(); // Unboxing (Extracting the value from Object)    Integer kRef = i; // Auto Boxing (Integer kRef = new Integer(i))  int l = kRef; // (Auto boxing)    }  } |

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ **Anonymous Class**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

# **Anonymous Inner Class in Java**

Prerequisites :- [Nested Classes in Java](https://www.geeksforgeeks.org/nested-classes-java/)  
It is an inner class without a name and for which only a single object is created. An anonymous inner class can be useful when making an instance of an object with certain “extras” such as overloading methods of a class or interface, without having to actually subclass a class.

Anonymous inner class are mainly created in two ways:

* Class (may be abstract or concrete)
* Interface

In the program, interface Age is created with getAge() method and x=21.  Myclass is written as implementation class of Age interface. As done in Program, there is no need to write a  separate class Myclass. Instead,   directly copy the code of Myclass into this parameter, as shown here:

|  |  |
| --- | --- |
|  |  |

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ **immutable Class** \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

|  |
| --- |
| **How to create Immutable class in Java?** An object is immutable if its state cannot change after construction. Immutable objects don’t expose any way for other objects to modify their state; the object’s fields are initialized only once inside the constructor and never change again. 1. Usage of Immutable Classes Nowadays, the “must-have” specification for every software application is to be distributed and multi-threaded—multi-threaded applications always cause headaches for developers since developers are required to protect the state of their objects from concurrent modifications of several threads at the same time, for this purpose, developers normally use the Synchronizedblocks whenever they modify the state of an object.  With immutable classes, states are never modified; every modification of a state results in a new instance, hence each thread would use a different instance and developers wouldn’t worry about concurrent modifications. 2. Some Popular Immutable Classes **String**is the most popular immutable class in Java. Once initialized its value cannot be modified. Operations like **trim(), substring(), replace()** always return a new instance and don’t affect the current instance, that’s why we usually call **trim()**as the following:  String alex = "Alex";  alex = alex.trim();  Another example from JDK is the wrapper classes like: **Integer, Float, Boolean** … these classes don’t modify their state, however they create a new instance each time you try to modify them.  Integer a =3;  a += 3;  After calling **a += 3,** a new instance is created holding the value: 6 and the first instance is lost. 3. How Do We Create an Immutable Class In order to create an immutable class, you should follow the below steps:   1. Make your class **final,**so that no other classes can extend it. 2. Make all your fields **final,**so that they’re initialized only once inside the constructor and never modified afterward. 3. Don’t expose setter methods. 4. When exposing methods which modify the state of the class, you must always return a new instance of the class. 5. If the class holds a mutable object:    * Inside the constructor, make sure to use a clone copy of the passed argument and never set your mutable field to the real instance passed through constructor, this is to prevent the clients who pass the object from modifying it afterwards.    * Make sure to always return a clone copy of the field and never return the real object instance. |

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ **Multi Thread** \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

# **Multithreading in Java**

Multithreading is a Java feature that allows concurrent execution of two or more parts of a program for maximum utilization of CPU. Each part of such program is called a thread. So, threads are light-weight processes within a process.  
  
Threads can be created by using two mechanisms :  
1. Extending the Thread class  
2. Implementing the Runnable Interface  
   
**Thread creation by extending the Thread class**  
We create a class that extends the **java.lang.Thread** class. This class overrides the run() method available in the Thread class. A thread begins its life inside run() method. We create an object of our new class and call start() method to start the execution of a thread. Start() invokes the run() method on the Thread object.

|  |
| --- |
| **package** multiThreading;  **public** **class** ThreadExample {  **public** **static** **void** main(String args[]) {  calculate obj = **new** calculate();  task ob1 = **new** task(obj);  task ob2 = **new** task(obj);  // All those 2 thread will not print in a mixed way since we use synchronized  // method  // To start Thread 1 method run()  ob1.start();  // To start Thread 2 method run()  ob2.start();  }  }  // One way to create Thread is by extending the Thread class  **class** task **extends** Thread {  calculate calc;  task(calculate calc) {  **this**.calc = calc;  }  **public** **void** run() {  calc.cal();  }  }  **class** calculate {  **public** **synchronized** **void** cal() {  **for** (**int** i = 0; i < 10; i++) {  System.***out***.println("Thread " + i);  **try** {  Thread.*sleep*(400);  } **catch** (Exception e) {  System.***out***.println(e);  }  }  }  } |

**Thread creation by implementing the Runnable Interface**  
We create a new class which implements java.lang.Runnable interface and override run() method. Then we instantiate a Thread object and call start() method on this object.

|  |
| --- |
| //the Runnable Interface  **class** MultithreadingDemo **implements** Runnable {  **public** **void** run() {  **try** {  // Displaying the thread that is running  System.***out***.println("Thread " + Thread.*currentThread*().getId() + " is running");  } **catch** (Exception e) {  // Throwing an exception  System.***out***.println("Exception is caught");  }  }  }  //Main Class  **class** Multithread {  **public** **static** **void** main(String[] args) {  **int** n = 8; // Number of threads  **for** (**int** i = 0; i < 8; i++) {  Thread object = **new** Thread(**new** MultithreadingDemo());  object.start();  }  }  } |

# **Synchronized in Java**

[Multi-threaded](http://quiz.geeksforgeeks.org/multithreading-in-java/)programs may often come to a situation where multiple threads try to access the same resources and finally produce erroneous and unforeseen results.

So it needs to be made sure by some synchronization method that only one thread can access the resource at a given point of time.

Java provides a way of creating threads and synchronizing their task by using synchronized blocks. Synchronized blocks in Java are marked with the synchronized keyword. A synchronized block in Java is synchronized on some object. All synchronized blocks synchronized on the same object can only have one thread executing inside them at a time. All other threads attempting to enter the synchronized block are blocked until the thread inside the synchronized block exits the block.

### **Types of Synchronization**

There are two types of synchronization

1. Process Synchronization

### Thread Synchronization

### **Thread Synchronization**

There are two types of thread synchronization mutual exclusive and inter-thread communication.

1. Mutual Exclusive
   1. Synchronized method.
   2. Synchronized block.
   3. static synchronization.

### **Mutual Exclusive**

Mutual Exclusive helps keep threads from interfering with one another while sharing data. This can be done by three ways in java:

1. by synchronized method
2. by synchronized block
3. by static synchronization

|  |
| --- |
| **Understanding the problem without Synchronization** In this example, there is no synchronization, so output is inconsistent. Let's see the example:   1. **class** Table{ 2. **void** printTable(**int** n){//method not synchronized 3. **for**(**int** i=1;i<=5;i++){ 4. System.out.println(n\*i); 5. **try**{ 6. Thread.sleep(400); 7. }**catch**(Exception e){System.out.println(e);} 8. } 10. } 11. } 13. **class** MyThread1 **extends** Thread{ 14. Table t; 15. MyThread1(Table t){ 16. **this**.t=t; 17. } 18. **public** **void** run(){ 19. t.printTable(5); 20. } 22. } 23. **class** MyThread2 **extends** Thread{ 24. Table t; 25. MyThread2(Table t){ 26. **this**.t=t; 27. } 28. **public** **void** run(){ 29. t.printTable(100); 30. } 31. } 33. **class** TestSynchronization1{ 34. **public** **static** **void** main(String args[]){ 35. Table obj = **new** Table();//only one object 36. MyThread1 t1=**new** MyThread1(obj); 37. MyThread2 t2=**new** MyThread2(obj); 38. t1.start(); 39. t2.start(); 40. } 41. }   Output: 5  100  10  200  15  300  20  400  25  500   **Java synchronized method** If you declare any method as synchronized, it is known as synchronized method.  Synchronized method is used to lock an object for any shared resource.  When a thread invokes a synchronized method, it automatically acquires the lock for that object and releases it when the thread completes its task.   1. //example of java synchronized method 2. **class** Table{ 3. **synchronized** **void** printTable(**int** n){//synchronized method 4. **for**(**int** i=1;i<=5;i++){ 5. System.out.println(n\*i); 6. **try**{ 7. Thread.sleep(400); 8. }**catch**(Exception e){System.out.println(e);} 9. } 11. } 12. } 14. **class** MyThread1 **extends** Thread{ 15. Table t; 16. MyThread1(Table t){ 17. **this**.t=t; 18. } 19. **public** **void** run(){ 20. t.printTable(5); 21. } 23. } 24. **class** MyThread2 **extends** Thread{ 25. Table t; 26. MyThread2(Table t){ 27. **this**.t=t; 28. } 29. **public** **void** run(){ 30. t.printTable(100); 31. } 32. } 34. **public** **class** TestSynchronization2{ 35. **public** **static** **void** main(String args[]){ 36. Table obj = **new** Table();//only one object 37. MyThread1 t1=**new** MyThread1(obj); 38. MyThread2 t2=**new** MyThread2(obj); 39. t1.start(); 40. t2.start(); 41. } 42. }   Output: 5  10  15  20  25  100  200  300  400  500 |

­­­­­­­\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ **Finally** \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

>> Used to place important code, it will be executed whether exception is handled or not.

|  |
| --- |
| class FinallyExample{  public static void main (String args[]){  try(  int = x 100;  )catch(Extraction e){System.out.println(e);}  finally  {System.out.println("Finally bloxk is executed")};  }  } |

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ **Finalize** \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

>> Used to perform clean up processing just before object is garbage collected.

|  |
| --- |
| public class FinalizeExample {  public void finalize()  {  System.out.println("Finallized is called");  }  public static void main (String args[]){  FinalizeExample f1 = new FinalizeExample();  FinalizeExample f2 = new FinalizeExample();    f1 = null;  f2 = null;  System.gc()  }  }  Consle Output >> Finallized is called  Finallized is called |

\_\_\_\_\_\_\_\_\_\_\_\_\_\_ **Difference between StringBuffer and StringBuilder** \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

|  |  |
| --- | --- |
| **StringBuffer** | **StringBuilder** |
| operations are thread-safe and synchronized | operations are not thread-safe |
| is to be used when multiple threads are working on same String | Is to be used in the single thread environment |
| performance is slower compared to StringBuilder | Performance is fast compared to stringBuffer |

|  |
| --- |
| public class performaceTest{    public static void main(String[] args){    int N = 999999999;  long t;  {  StringBuffer sb = new StringBuffer();  t = System.currentTimeMillis();  for (int i = N; i --> 0;){  sb.append("");  }  System.out.println(System.currentTimeMillis() - t);  }    {  StringBuilder sb = new StringBuilder();  t = System.currentTimeMillis();  for (int i = N; i --> 0;){  sb.append("");  }  System.out.println(System.currentTimeMillis() - t);  }  }  } |

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ **Difference between Heap & Stack Memory** \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

|  |  |  |
| --- | --- | --- |
|  | **Stack** | **Heap** |
| **Memory** | Stack memory is use only by one thread of execution | heap memory used by all the parts of the application |
| **Access** | stack memory can't be access by other thread | Object stored in the heap are globally accessible |
| **Memory**  **management** | Follows LIFO manner to free memory | memory management based on generation associated in each object |
| **Lifetime** | Exists until the end of execution of the tread, | Heap memory lies from the stat till the end of application execution |
| **Usage** | Stack memory only contain local primitive variables | whenever an object created, it stored in the Heap space and reference variables to objects in heap space |

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**The difference between equals()and == operator?**\_\_\_\_\_\_\_\_\_\_\_\_

|  |  |
| --- | --- |
| **Equals() method** | **== equality operator** |
| used for checking equality of two objects defined by business logic | used to compare primitives and objects |

**OOP Questions:**

\_\_\_\_\_\_\_\_\_**The difference between Abstract class and Interfaces**\_\_\_\_\_\_\_\_\_\_\_\_

|  |  |
| --- | --- |
| **Abstract Class** | **Interfaces** |
| An abstract class can provide complete default code and/ or just the details that have to be overridden. | An interface can’t provide any code at all, just the signature. |
| In case of abstract class, a class may extend only one abstract class. | A class may implement several interfaces. |
| An abstract class can have non-abstract methods. | All methods of an interface are abstract |
| An abstract class can have interface variable | An interface visibility must be public or none. |
| An abstract class can contain constructors | An interface can’t contain constructors |
| Abstract classes are fast | Interfaces are slow as it requires extra indirection to find corresponding method in the actual class. |

|  |  |
| --- | --- |
| **Runtime Polymorphism**  // If we add abstract to the parent class, we can't create object, but runtime environment create object, so we must create constructor in order to inherit the object.  **abstract** **class** shape{  shape()  {  System.***out***.println("Shape is constructed");  }  **abstract** **void** draw(); // No definition needed  **void** draw1(); // No definition  }  /////////////// Child Class inherit Parent ///////////    // Object to object inheritance  **class** Circle **extends** shape{  **void** draw() // Overriding  {  System.***out***.println("Drawing a Circle");  }  }  **class** Rectangle **extends** shape{  **void** draw()  {  System.***out***.println("Drawing a Rectangle");  }  }  **class** Polygon **extends** shape{  **void** draw()  {  System.***out***.println("Drawing Polygon");  }  }    ////////////////// Driver Class /////////////////////  **public** **class** InterviewQuestion{    **public** **static** **void** main (String [] args)  {  // Run time polymorphism  // The parent class point to the child  shape s;  s = **new** Circle();  s.draw();    s = **new** Rectangle();  s.draw();    s = **new** Polygon();  s.draw();  }  } | **Runtime Polymorphism**  // Interface is not object to object inheritance. We or runtime environment we can't create object  // By default all methods are abstract.  **interface** shape{  **void** draw(); // by default it is public & abstract  }  ////// No longer be Parent Child relationship ////////    // This is no longer be object to object inheritance  **class** Circle **implements** shape{  **void** draw() // Not Overriding  {  System.***out***.println("Drawing a Circle");  }  }  **class** Rectangle **implements** shape{  **void** draw()  {  System.***out***.println("Drawing a Rectangle");  }  **class** Polygon **implements** shape{  **void** draw()  {  System.***out***.println("Drawing a Polygon");  }  }    ////////////////// Driver Class /////////////////////  **public** **class** InterviewQuestion{    **public** **static** **void** main (String [] args)  {  // Run time polymorphism  // The parent class point to the child  shape s;  s = **new** Circle();  s.draw();    s = **new** Rectangle();  s.draw();  s = **new** Polygon();  s.draw();  }  } |

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ **Polymorphism** \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

>> Polymorphism is the ability of an object to take on many forms (One interface many implementations)

>> There are two types of Polymorphism: Runtime (dynamic method dispatch) and Compiletime.

>> Runtime Polymorphism is parent child inheritance form (a call to an overridden method is resolved at run time rather

than compile time, it is a process of an overridden method called through the reference variable of super class)

\_\_\_\_\_\_\_\_\_\_\_\_\_ **Difference between Overloading & Overridding method** \_\_\_\_\_\_\_\_\_\_\_\_\_\_

>> You can’t override a Static or Private method in java.

|  |  |  |
| --- | --- | --- |
|  | **Overload** | Overriding |
| **Definition** | Methods of the same class shares the same name, but each method must have different number of parameters or parameters having different types & order. | Sub class have the same method with same name and exactly the same number and type of parameters and same return type as a super class. |
| **Behavior** | To add or extend more to method’s behavior. | To change existing behavior of method. |
| **Polymorphism** | It is compile time polymorphism | It is run time polymorphism |
| **Signature** | Methods must have different signature | Methods must have same signature |
| **Inheritance** | It may or may not need inheritance in m | It always requires inheritance in method |

|  |  |
| --- | --- |
| **Overloading**  **class** overloadingExample{    **static** **int** add(**int** a,**int** b)  {  **return** a+b;  }  **static** **double** add(**double** a, **double** b)  {  **return** a+b;  }  }  **class** InterviewQuestion{  **public** **static** **void** main(String[] args) {  // if the overloading method is Static, you can call it  // without creating object System.***out***.println(overloadingExample.*add*(5, 5));  System.***out***.println(overloadingExample.*add*(5.2, 5.2));  }  }  >> Counsel Output: 10  10.4 | **Overriding**  **class** Animal {  **void** eat()  {  System.***out***.println("eating...");  }  }  **class** Dog **extends** Animal {  **void** eat()  {  System.***out***.println("eating bread...");  }  }  **class** InterviewQuestion{  **public** **static** **void** main(String[] args) {  Animal s = **new** Animal();  s.eat();  s = **new** Dog();  s.eat();  }  }  >> Counsel Output: eating...  eating bread... |

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ **Regex** \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

## **Regular expressions in Java**

A regular expression, or regex, is a sequence of characters that specifies a pattern which can be searched for in a text. A regex defines a set of strings, usually united for a given purpose. Suppose you need a way to formalize and refer to all the strings that make up the format of an email address. Since there are a near infinite number of possible email addresses, it’d be hard to enumerate them all. However, as we know an email address has a specific structure, and we can encode that using the regex syntax.

A regex processor translates a regular expression into an internal representation which can be executed and matched against the text being searched. It will tell you whether a string is in the set of strings defined by a pattern or find a substring that belongs in that set.

## **Useful Java classes & methods**

Most languages have a regular expressions implementation either baked in or provided by a library. Java is no exception. Below are the classes you have to know in order to be effective using Java Regex.

### Pattern

[**Pattern**](https://docs.oracle.com/javase/8/docs/api/java/util/regex/Pattern.html) is a compiled representation of a regular expression. Below is the list of the most frequently used methods in the Pattern class API.

Pattern compile(String regex) – Compiles the given regular expression into a pattern.

Pattern compile(String regex, int flags) – compiles the given regular expression into a pattern with the given flags.

boolean matches(String regex) – returns whether or not this string matches the given regular expression.

String[] split(CharSequence input) – splits the given input sequence around matches of this pattern.

String quote(String s) – returns a literal pattern String for the specified String s.

Predicate asPredicate() – creates a predicate which can be used to match a string.

### Matcher

A [**matcher**](https://docs.oracle.com/javase/8/docs/api/java/util/regex/Matcher.html) is the engine that performs match operations on a character sequence by interpreting a Pattern. Below is the list of the most frequently used methods in the Matcher class API.

boolean matches() – attempts to match the entire region against the pattern.

boolean find() – attempts to find the next subsequence of the input that matches the pattern.

int start() – returns the start index of the last match.

int end() – returns the offset after the last character matched.

By compiling a pattern, obtaining a matcher for it, you can match many texts for the pattern efficiently. So if you expect to process lots of texts, compile a matcher, cache it and use it repeatedly.

## **Java Regex syntax**

Let’s move on now to the syntax for Java Regex. The Pattern.compile method takes a String, which is the regex that defines a set of matching strings. Naturally, it has to have a tricky syntax, otherwise a single string defining the pattern can only represent itself.

A regular character in the regex syntax matches that character in the text. If you’ll create a Pattern with Pattern.compile("a") it will only match only the String “a”. There is also an escape character, which is the backslash “\”. It is used to distinguish when the pattern contains an instruction in the syntax or a character. Let’s look at an example as to why we need an escape character. Imagine “[” has a special meaning in the regular expression syntax (it has). How can you determine if “[” is a command to the matching engine or a pattern containing only the bracket? You cannot, so to specify the characters that are also the commands in the syntax you need to escape them. It means “\[” is a pattern for the string “[“, and “[” is part of a command. What about trying to match a backslash? You need to escape it too, so be prepared to see something like “\\\\” in the regex code.

## **Character classes**

On top of specifying the expressions that contain individual characters only, you can define the whole classes of characters. Think of them as sets, if a character in some text belongs to the character class, it is matched.

Here is a table with the most used character classes.

* [abc] simple, matches a or b, or c.
* [\^abc] negation, matches everything except a, b, or c.
* [a-c] range, matches a or b, or c.
* [a-c[f-h]] union, matches a, b, c, f, g, h.
* [a-c&&[b-c]] intersection, matches b or c.
* [a-c&&[\^b-c]] subtraction, matches only a.

<h2<="" h2="" style="box-sizing: inherit; caret-color: rgb(51, 51, 51); color: rgb(51, 51, 51); font-family: "Open Sans"; font-size: 16px; font-style: normal; font-variant-caps: normal; font-weight: normal; letter-spacing: normal; orphans: auto; text-align: start; text-indent: 0px; text-transform: none; white-space: normal; widows: auto; word-spacing: 0px; -webkit-text-size-adjust: auto; -webkit-text-stroke-width: 0px; text-decoration: none;">

For your convenience, there are some useful classes defined already. For example, digits are a perfect example of a useful character class. For example a 5 digit number could be coded into a pattern as “[0-9][0-9][0-9][0-9][0-9]”, but it’s quite ugly. So there’s a shorthand for that: “\d”.

Here are the other classes you need to know.

* . Any character
* \d A digit: [0-9]
* \D A non-digit: [\^0-9]
* \s A whitespace character: [ \t\n\x0B\f\r]
* \S A non-whitespace character: [\^\s]
* \w A word character: [a-zA-Z\_0-9]
* \W A non-word character: [\^\w]

Note that the letter specifying a predefined character class is typically lowercase, the uppercase version tends to mean the negation of the class.

Also, note that a dot “.” is a character class, which contains all the characters. Particularly useful, but remember to escape it when you need to match the actual dot character.

## **Boundary matchers**

Next, there’s syntax to specify the position of the matched sequence in the original text you’re searching. If you only need to filter out the strings that start with an email address or something, this is extremely useful.

* ^ The beginning of a line.
* $ The end of a line.
* \b A word boundary.
* \B A non-word boundary.
* \A The beginning of the input.
* \G The end of the previous match.
* \Z The end of the input but for the final terminator, if any.
* \z The end of the input.

A noteworthy combination of the boundary matchers is the “^pattern$” which will only match the text if it is the full pattern.

## **Logical operations**

Now we’re getting into more advanced territory. If a pattern is more than a single character long, it will match a longer string too. In general “XY” in the regex syntax matches X followed by Y. However, there’s also an OR operation, denoted by the post “|”. The “X|Y” regex means it is either X or Y. This is a very powerful feature; you can combine the character classes or sequences of characters (include them in brackets).

## **Quantifiers**

On top of everything, you can say how many times the sequence of characters can be repeated for the match. The regex “1” only matches the input “1”, but if we need to match a string of any length consisting of the character “1” you need to use one of the following quantifiers.

* \* matches zero or more occurrences.
* + matches one or more occurrences.
* ? matches zero or one occurrence.

## **Groups and backreferences**

A group is a captured subsequence of characters which may be used later in the expression with a backreference. We’ve mentioned already that if you enclose a group of characters in parentheses, you can apply quantifiers or logical or to the whole group. What is even more awesome is that you can refer to the actual characters in the text matched by the group, later.  
Here’s how you do it:  
(...) – defines a group, in the parantheses  
\N – refers to a matched group  
For example:  
(\d\d) – a group of two digits  
(\d\d)/\1– two digits repeated twice, \1 – refers to the matched group, so this regular expression matches the strings that look like “aabb” where a and b are both digits.

## **Pattern flags**

Remember when we talked about the useful API in Java for the regular expressions, there was a method to compile a pattern that took the flags. These will control how the pattern behaves.

Here are some flags that can be useful here and there.

* Pattern.CASE\_INSENSITIVE – enables case-insensitive matching
* Pattern.COMMENTS – whitespace and comments starting with # are ignored until the end of a line.
* Pattern.MULTILINE – one expression can match multiple lines.
* Pattern.DOTALL – the expression “.” matches any character, including a line terminator
* Pattern.UNIX\_LINES – only the ‘\n’ line terminator is recognized in the behavior of ., ^, and $.



**java.util.regex.Pattern class:**

**1) Pattern.matches()**

We have already seen the usage of this method in the above example where we performed the search for string “book” in a given text. This is one of simplest and easiest way of searching a String in a text using Regex.

String content = "This is a tutorial Website!";

String patternString = ".\*tutorial.\*";

boolean isMatch = Pattern.matches(patternString, content);

System.out.println("The text contains 'tutorial'? " + isMatch);

As you can see we have used matches() method of Pattern class to search the pattern in the given text. The pattern .\*tutorial.\* allows zero or more characters at the beginning and end of the String “tutorial” (the expression .\* is used for zero and more characters).

**Limitations**: This way we can search a single occurrence of a pattern in a text. For matching multiple occurrences you should use the Pattern.compile() method (discussed in the next section).

**2) Pattern.compile()**

In the above example we searched a string “tutorial” in the text, that is a case sensitive search, however if you want to do a CASE INSENSITIVE search or want to do search multiple occurrences then you may need to first compile the pattern using Pattern.compile() before searching it in text. This is how this method can be used for this case.

String content = "This is a tutorial Website!";

String patternString = ".\*tuToRiAl.";

Pattern pattern = Pattern.compile(patternString, Pattern.CASE\_INSENSITIVE);

Here we have used a flag Pattern.CASE\_INSENSITIVE for case insensitive search, there are several other flags that can be used for different-2 purposes. To read more about such flags [refer this document](https://docs.oracle.com/javase/tutorial/essential/regex/pattern.html).

**Now what**: We have obtained a Pattern instance but how to match it? For that we would be needing a Matcher instance, which we can get using Pattern.matcher() method. Lets discuss it.

**3) Pattern.matcher() method**

In the above section we learnt how to get a Pattern instance using compile() method. Here we will learn How to get Matcher instance from Pattern instance by using matcher() method.

String content = "This is a tutorial Website!";

String patternString = ".\*tuToRiAl.\*";

Pattern pattern = Pattern.compile(patternString, Pattern.CASE\_INSENSITIVE);

Matcher matcher = pattern.matcher(content);

boolean isMatched = matcher.matches();

System.out.println("Is it a Match?" + isMatched);

Output:

Is it a Match?true

**4) Pattern.split()**

To split a text into multiple strings based on a delimiter (Here delimiter would be specified using **regex**), we can use Pattern.split() method. This is how it can be done.

import java.util.regex.\*;

class RegexExample2{

public static void main(String args[]){

String text = "ThisIsChaitanya.ItISMyWebsite";

// Pattern for delimiter

String patternString = "is";

Pattern pattern = Pattern.compile(patternString, Pattern.CASE\_INSENSITIVE);

String[] myStrings = pattern.split(text);

for(String temp: myStrings){

System.out.println(temp);

}

System.out.println("Number of split strings: "+myStrings.length);

}}

Output:

Th

Chaitanya.It

MyWebsite

Number of split strings: 4

The second split String is null in the output.

**java.util.regex.Matcher Class**

We already discussed little bit about Matcher class above. Lets recall few things:

**Creating a Matcher instance**

String content = "Some text";

String patternString = ".\*somestring.\*";

Pattern pattern = Pattern.compile(patternString);

Matcher matcher = pattern.matcher(content);

**Main methods**

**matches()**: It matches the regular expression against the whole text passed to the Pattern.matcher() method while creating Matcher instance.

...

Matcher matcher = pattern.matcher(content);

boolean isMatch = matcher.matches();

**lookingAt()**: Similar to matches() method except that it matches the regular expression only against the beginning of the text, while matches() search in the whole text.

**find()**: Searches the occurrences of of the regular expressions in the text. Mainly used when we are searching for multiple occurrences.

**start() and end()**: Both these methods are generally used along with the find() method. They are used for getting the start and end indexes of a match that is being found using find() method.

**Lets take an example to find out the multiple occurrences using Matcher methods:**

package beginnersbook.com;

import java.util.regex.\*;

class RegexExampleMatcher{

public static void main(String args[]){

String content = "ZZZ AA PP AA QQQ AAA ZZ";

String string = "AA";

Pattern pattern = Pattern.compile(string);

Matcher matcher = pattern.matcher(content);

while(matcher.find()) {

System.out.println("Found at: "+ matcher.start()

+

" - " + matcher.end());

}

}

}

Output:

Found at: 4 - 6

Found at: 10 - 12

Found at: 17 - 19

Now we are familiar with Pattern and Matcher class and the process of matching a regular expression against the text. Lets see what kind of various options we have to define a regular expression:

**1) String Literals**

Lets say you just want to search a particular string in the text for e.g. “abc” then we can simply write the code like this: Here text and regex both are same.  
Pattern.matches("abc", "abc")

**2) Character Classes**

A character class matches a single character in the input text against multiple allowed characters in the character class. For example [Cc]haitanya would match all the occurrences of String “chaitanya” with either lower case or upper case C”. Few more examples:  
Pattern.matches("[pqr]", "abcd"); It would give false as no p,q or r in the text  
Pattern.matches("[pqr]", "r"); Return true as r is found  
Pattern.matches("[pqr]", "pq"); Return false as any one of them can be in text not both.

Here is the complete list of various character classes constructs:  
[abc]: It would match with text if the text is having either one of them(a,b or c) and only once.  
[^abc]:  Any single character except a, b, or c (^ denote negation)  
[a-zA-Z]:  a through z, or A through Z, inclusive (range)  
[a-d[m-p]]:  a through d, or m through p: [a-dm-p] (union)  
[a-z&&[def]]:  Any one of them (d, e, or f)  
[a-z&&[^bc]]: a through z, except for b and c: [ad-z] (subtraction)  
[a-z&&[^m-p]]:  a through z, and not m through p: [a-lq-z] (subtraction)

**Predefined Character Classes – Metacharacters**

These are like short codes which you can use while writing regex.

Construct Description

. -> Any character (may or may not match line terminators)

\d -> A digit: [0-9]

\D -> A non-digit: [^0-9]

\s -> A whitespace character: [ \t\n\x0B\f\r]

\S -> A non-whitespace character: [^\s]

\w -> A word character: [a-zA-Z\_0-9]

\W -> A non-word character: [^\w]

For e.g.  
Pattern.matches("\\d", "1"); would return true  
Pattern.matches("\\D", "z"); return true  
Pattern.matches(".p", "qp"); return true, dot(.) represent any character

**Boundary Matchers**

^ Matches the beginning of a line.

$ Matches then end of a line.

\b Matches a word boundary.

\B Matches a non-word boundary.

\A Matches the beginning of the input text.

\G Matches the end of the previous match

\Z Matches the end of the input text except the final terminator if any.

\z Matches the end of the input text.

For e.g.  
Pattern.matches("^Hello$", "Hello"): return true, Begins and ends with Hello  
Pattern.matches("^Hello$", "Namaste! Hello"): return false, does not begin with Hello  
Pattern.matches("^Hello$", "Hello Namaste!"): return false, Does not end with Hello

**Quantifiers**

Greedy Reluctant Possessive Matches

X? X?? X?+ Matches X once, or not at all (0 or 1 time).

X\* X\*? X\*+ Matches X zero or more times.

X+ X+? X++ Matches X one or more times.

X{n} X{n}? X{n}+ Matches X exactly n times.

X{n,} X{n,}? X{n,}+ Matches X at least n times.

X{n, m) X{n, m)? X{n, m)+ Matches X at least n time, but at most m times.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_­­­\_\_ **String (The Unique Class)** \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

To convert any object to string there are several methods in Java

String convertedToString = String.valueOf(Object); //method 1

String convertedToString = "" + Object; //method 2

String convertedToString = Object.toString(); //method 3

|  |
| --- |
| ///////////////////////////////////// String to Primitive //////////////////////////////////////////      String st = "1112";      // parse()  **int** is = Integer.*parseInt*(st); // Output 1112  **double** ds = Double.*parseDouble*(st); // Output 1112 or 1112.2 (if input Sting is decimal)  **boolean** bs = Boolean.*parseBoolean*(st); // Output true (if input String is true)      // valueOf()  **int** ins = Integer.*valueOf*(st); // Output 1112  **double** dos = Double.*valueOf*(st); // Output 1112.0 or 1112.2 (if input Sting is decimal)  **boolean** bos = Boolean.*valueOf*(st); // Output true (if input String is true)      // charAt()  **char** cs = st.charAt(0); // Output 1 (at specified index)    // toCharArrayt()  **char**[] cha = st.toCharArray(); // Output 1112      //////////////////////////////////// Primitive to String //////////////////////////////////////////    **int** in = 5;  **double** db = 5.5;  **char** ch = 'a';  **boolean** bl = **true**;    // toString()  String sti = Integer.*toString*(in); // Output 5  String std = Double.*toString*(db); // Output 5.5  String stc = Character.*toString*(ch); // Output a  String stb = Boolean.*toString*(bl); // Output true    // valueOf()  String stin = String.*valueOf*(in); // Output 5  String stdb = String.*valueOf*(db); // Output 5.5  String stch = String.*valueOf*(ch); // Output a  String stbl = String.*valueOf*(bl); // Output true      //////////////////////////////////// Primitive Array to String /////////////////////////////////////    **int**[] arr = {1, 2, 3};  String[] st = {"a", "b"};  **char**[] ch = {'a', 'b'};  **boolean**[] bl = {**true**, **false**};    // Arrays.toString()  String sti = Arrays.*toString*(arr); // Output [1,2,3]  String sts = Arrays.*toString*(st); // Output [a, b]  String stc = Arrays.*toString*(ch); // Output [a, b]  String stb = Arrays.*toString*(bl); // Output [true, false]      // valueOf()  String stch = String.*valueOf*(ch); // Output ab |

**String Methods**

1. **int length():**Returns the number of characters in the String.

"GeeksforGeeks".length();  // returns 13

1. [**Char charAt(int i)**](https://www.geeksforgeeks.org/java-string-charat-method-example/)**:**Returns the character at ith index.

"GeeksforGeeks".charAt(3); // returns  ‘k’

1. [**String substring (int i)**](https://www.geeksforgeeks.org/substring-in-java/)**:**Return the substring from the ith index character to end.

"GeeksforGeeks".substring(3); // returns “ksforGeeks”

1. [**String substring (int i, int j)**](https://www.geeksforgeeks.org/substring-in-java/)**:**Returns the substring from i to j-1 index.

"GeeksforGeeks".substring(2, 5); // returns “eks”

1. [**String concat( String str)**](https://www.geeksforgeeks.org/java-string-concat-examples/)**:**Concatenates specified string to the end of this string.

String s1 = ”Geeks”;

String s2 = ”forGeeks”;

String output = s1.concat(s2); // returns “GeeksforGeeks”

1. [**int indexOf (String s)**](https://www.geeksforgeeks.org/java-string-indexof/)**:**Returns the index within the string of the first occurrence of the specified string.

String s = ”Learn Share Learn”;

int output = s.indexOf(“Share”); // returns 6

1. [**int indexOf (String s, int i)**](https://www.geeksforgeeks.org/java-string-indexof/)**:**Returns the index within the string of the first occurrence of the specified string, starting at the specified index.

String s = ”Learn Share Learn”;

int output = s.indexOf(‘a’,3);// returns 8

1. [**Int lastIndexOf( String s)**](https://www.geeksforgeeks.org/java-lang-string-lastindexof-method/)**:**Returns the index within the string of the last occurrence of the specified string.

String s = ”Learn Share Learn”;

int output = s.lastIndexOf(‘a’); // returns 14

1. **boolean equals( Object otherObj):**Compares this string to the specified object.

Boolean out = “Geeks”.equals(“Geeks”); // returns true

Boolean out = “Geeks”.equals(“geeks”); // returns false

1. [**boolean  equalsIgnoreCase (String anotherString)**](https://www.geeksforgeeks.org/equalsignorecase-in-java/)**:**Compares string to another string, ignoring case considerations.
2. Boolean out= “Geeks”.equalsIgnoreCase(“Geeks”); // returns true

Boolean out = “Geeks”.equalsIgnoreCase(“geeks”); // returns true

1. [**int compareTo( String anotherString)**](https://www.geeksforgeeks.org/java-lang-string-compareto/)**:**Compares two string lexicographically.

int out = s1.compareTo(s2);  // where s1 ans s2 are

// strings to be compared

This returns difference s1-s2. If :

out < 0 // s1 comes before s2

out = 0 // s1 and s2 are equal.

out > 0 // s1 comes after s2.

1. **int compareToIgnoreCase( String anotherString):**Compares two string lexicographically, ignoring case considerations.

int out = s1.compareToIgnoreCase(s2);

// where s1 ans s2 are

// strings to be compared

This returns difference s1-s2. If :

out < 0 // s1 comes before s2

out = 0 // s1 and s2 are equal.

out > 0 // s1 comes after s2.

Note- In this case, it will not consider case of a letter (it will ignore whether it is uppercase or lowercase).

1. [**String toLowerCase()**](https://www.geeksforgeeks.org/java-string-tolowercase-examples/)**:**Converts all the characters in the String to lower case.

String word1 = “HeLLo”;

String word3 = word1.toLowerCase(); // returns “hello"

1. [**String toUpperCase()**](https://www.geeksforgeeks.org/java-touppercase-examples/)**:**Converts all the characters in the String to upper case.

String word1 = “HeLLo”;

String word2 = word1.toUpperCase(); // returns “HELLO”

1. [**String trim()**](https://www.geeksforgeeks.org/java-string-trim-method-example/)**:**Returns the copy of the String, by removing whitespaces at both ends. It does not affect whitespaces in the middle.

String word1 = “ Learn Share Learn “;

String word2 = word1.trim(); // returns “Learn Share Learn”

1. [**String replace (char oldChar, char newChar)**](https://www.geeksforgeeks.org/java-lang-string-replace-method-java/)**:**Returns new string by replacing all occurrences of oldCharwith newChar.

String s1 = “feeksforfeeks“;

String s2 = “feeksforfeeks”.replace(‘f’ ,’g’); // returns “geeksgorgeeks”

|  |  |  |
| --- | --- | --- |
| |  | | --- | | **Client Side**  **HTML to handle the Structure**  **CSS to handel the Style**  **JavaScript to handle the behavior** |  |  | | --- | | **Server Side**  Web  S  er  ver  Application Server      Pojos  Model  Servlet  Controller | |

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ **Servlet** \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**>>** Java Servlet is server side technologies to extend the capability of web servers by providing support for dynamic response and data persistence.

>> Tha javax.servlet and javax.servlet.http packages provides interface and classes for writing our own servlets.

>> All servlets must implement the javax.servlet.Servlet interface, which defines servlet lifecycle method.

>> As most web applications are accessed by HTTP protocol, we mostly extend HttpServlet class. Servlet API hierarchy.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ **Difference between Get & Post method** \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

|  |  |
| --- | --- |
| **Get** | **Post** |
| Limited amount of data can be sent because data is sent in header | Large amount of data can be sent because data is sent in body |
| Not secured because data is exposed in URL bar | Secured because data is not exposed in URL bar |
| Can be bookmarked | Can’t bookmarked |
| Idempotent | Non idempotent |
| It is more efficient & used than post | It is less efficient & used |

>> **Session:** is a conversational state between client and server and it can consists of multiple request and

response client and server.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ **Different methods of Session Management** \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

|  |
| --- |
| **Session management API**  **Session Management**  **HTML hidden Field**  **Cookies**  **URL writing**  **User Authentication** |

>> Snice HTTP and web server both are stateless, the only way to maintain a session is when some unique

information about the session (session id) is passed between server & client in every request & response.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_ **Difference between ServletContext vs ServletConfig** \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

|  |  |
| --- | --- |
| **ServletConfig** | **ServletContext** |
| Servlet config object represent single servlet | It represents whole web application running on particular JVM & common for all the servlet |
| Its like local parameter associated with particular servlet | Its like global parameter associated with whole application |
| It’s a name value pair defined inside the servlet section of web.xml file so it has servlet wide scope | Servlet context has application wide scope so define outside of servlet tag in web.xml file |
| getServletConfig() method is used to get the config object | getServletContext() method is used to get the context object |
| E.g Shopping cart of a user is a specific to particular user so here we can use servlet config | To get the MIME type of a file or application session related information is stored using servlet context object |

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ **Life-cycle of a Servlet** \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

|  |
| --- |
| **Instantiation**  **Create Servlet interfaces**  **Loading**  **Load Servlet Class**  **Initialization**  **CALL init()**  **Destroy**  **Call destroy()**  **Request**  **Call service()** |

>> The entire life cycle of a Servlet is managed by the **Servlet container** which

uses the **javax.servlet.Servlet** interface.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ **Request Dispatcher** \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

>> Request dispatcher interface is used to forward the request to another resource that can be HTML, jsp

Or another servlet in same application.

>> There are two methods defined in this interface: 1, void forward(), & 2, include()

**1, forward():**

|  |
| --- |
| Request. forward()  **Servlet1**  **Servlet2**  **Client** |

2, **Include():**

Request include()

**Servlet1**

**Servlet2**

**Client**

Response of Servlet2

Include

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ **How does Cookies work in Servlets?** \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

>> Cookies are text data sent by server to the client and it gets saved at the client local machine.

>> Servlet API provides cookies support through **javax.servlet.http.Cookie** class that implements Serializable &

cloneable interfaces.

>> **HttpServletRequest getCokies()** method is provided to get the array of Cookies form request, since there is

no point of adding Cookie to request, there are no methods to set or add cookie to request.

>> Similarly **HttpsServletReqponse addCookie(Cookie c)** method is provided to attach cookie in response

header, there are no getter methods for cookie.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ **MVC** \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

MVC (Model View Controller) Component Model:

MVC contains: Controller(To get the request and decide what to accomplish), Mode(use components to job

done), View ( received implementation from Controller & connect with CHTML,CSS,& javaScript

& send it to client)

MVC also connect with data base .

|  |  |  |
| --- | --- | --- |
| |  | | --- | | **Client Side**  **HTML to handle the Structure**  **CSS to handel the Style**  **JavaScript to handle the behavior** |  |  | | --- | | **Server Side**  Web  S  ERve  r  Application Server  MVC    Pojos  Model  Jsp  View  Servlet  Controler  Data  Server  DB | |

**JDBC (Java Database Connectivity):**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ **What is JDBC Driver?** \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

>> JDBC Driver is a software component that enables java application to interact

with the database.



>> There are four types of JDBC drivers: 1, **JDBC-ODBC bridge driver** 2, **Native-API**

**driver(partially java driver)**,3, **Network Protocol driver(fully java driver)**,

4, **Thin driver (fully java driver)**

\_\_\_\_\_\_\_\_\_\_\_\_ **What are the steps to connect to the database in java?** \_\_\_\_\_\_\_\_\_\_\_\_\_

>> There are 5 steps need to follow to connect to a database in java

1, **Registering the driver class** 2, **Creating connection** 3, **Creating statement**

4, **Executing quires** 5, **Closing connection**

<http://www.vogella.com/tutorials/MySQLJava/article.html>

|  |
| --- |
| **import** java.sql.Connection;  **import** java.sql.DriverManager;  **import** java.sql.SQLException;  **import** java.sql.Statement;  **class** InterviewQuestion {  **public** **static** **void** main(String[] args) {  **try** {  // This will load the MySQL driver, each DB has its own driver  Class.*forName*("oracle.jdbc.driver.OracleDriver");  // Setup the connection with the DB  Connection con = DriverManager.*getConnection*("url:jdvc:oracle:thin:@<hostname>:  <port num>:<DB name>","user","password: password");  // Statements allow to issue SQL queries to the database  Statement stmt = con.createStatement();  System.***out***.println("Created DB connectio..........");    } **catch** (ClassNotFoundException e)  {  // Todo auto generated catch block  e.printStackTrace();  }**catch** (SQLException e)  {  // Todo auto generated catch block  e.printStackTrace();  }  }  } |

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ **What are the difference JDBC API components?** \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

<https://www.javatpoint.com/jdbc-interview-questions>

>> **Interfaces:**

* **Connection:** The Connection object is created by using getConnection() method of DriverManager class. DriverManager is the factory for connection.
* **Statement:** The Statement object is created by using createStatement() method of Connection class. The Connection interface is the factory for Statement.
* **PreparedStatement:** The PrepareStatement object is created by using prepareStatement() method of Connection class. It is used to execute the parameterized query.
* **ResultSet:** The object of ResultSet maintains a cursor pointing to a row of a table. Initially, cursor points before the first row. The executeQuery() method of Statement interface returns the ResultSet object.
* **ResultSetMetaData:** The object of ResultSetMetaData interface cotains the information about the data (table) such as numer of columns, column name, column type, etc. The getMetaData() method of ResultSet returns the object of ResultSetMetaData.
* **DatabaseMetaData:** DatabaseMetaData interface provides methods to get metadata of a database such as the database product name, database product version, driver name, name of the total number of tables, the name of the total number of views, etc. The getMetaData() method of Connection interface returns the object of DatabaseMetaData.
* **CallableStatement:** CallableStatement interface is used to call the stored procedures and functions. We can have business logic on the database through the use of stored procedures and functions that will make the performance better because these are precompiled. The prepareCall() method of Connection interface returns the instance of CallableStatement.

>> **Classes:**

* **DriverManager:** The DriverManager class acts as an interface between the user and drivers. It keeps track of the drivers that are available and handles establishing a connection between a database and the appropriate driver. It contains several methods to keep the interaction between the user and drivers.
* **Blob:** Blob stands for the binary large object. It represents a collection of binary data stored as a single entity in the database management system.
* **Clob:** Clob stands for Character large object. It is a data type that is used by various database management systems to store character files. It is similar to Blob except for the difference that BLOB represent binary data such as images, audio and video files, etc. whereas Clob represents character stream data such as character files, etc.
* **SQLException** It is an Exception class which provides information on database access errors.
* **Type** the class that defines the constants that are used to identify generic SQL types, called JDBC types.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ **What is a JDBC connection Interface?** \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

>> The connection interface maintains a session with the database. It can be used for transaction management.

**Connection interface**

**1) public Statement createStatement():** creates a statement object that can be used to execute SQL

queries.

**2) public Statement createStatement(int resultSetType,int resultSetConcurrency):** Creates a

Statement object that will generate ResultSet objects with the given type and concurrency.

**3) public void setAutoCommit(boolean status):** is used to set the commit status.By default it is true.

**4) public void commit():** saves the changes made since the previous commit/rollback permanent.

**5) public void rollback():** Drops all changes made since the previous commit/rollback.

**6) public void close():** closes the connection and Releases a JDBC resources immediately.

\_\_\_\_\_ **What is the difference between execute, executeQuery**, **executeUpdate?** \_\_\_\_\_\_

>> execute(): This method can be use for any kind of SQL statement. This method returns Boolean value. TRUE indicates that query returned a ResultSet object and FALSE

|  |  |
| --- | --- |
| **execute()** | **executeQuery()** |
| Is used for any kind of SQL statement. This method returns Boolean value TRUE indicates that query returned a ResultSet object and FALSE indicates that query returned an int value or returned nothing. | Is used to execute the SQL statements which retrieves some data from the database. It returns a ResultSet object which contains the result returned by the query. It only used to execute only SELECT queries. |
|  |  |
| **executeUpdate()** | |
| Is used to execute the SQL statements which update or modify the database. It returns un int value which represents the number of rows affected by the query. This value will be the o for the statements which return nothing. This method only used to execute only non-SELECT queries. Ex DML: INSERT, UPDATE & DELETE. Ex DML: CREATE, ALTER | |

**Spring:**

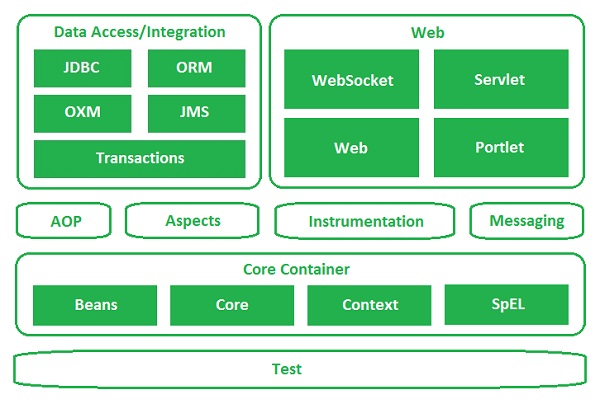
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ **What is Spring** \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

<https://www.javatpoint.com/spring-tutorial>

>> Spring is essentially a lightweight, integrated framework that can be used for developing enterprise applications. It is a

framework of framework.

\_\_\_\_\_\_\_\_\_\_\_ **What are the difference modules of the Spring framework?** \_\_\_\_\_\_\_\_\_\_\_\_



>> Sprig Context – for dependency injection

>> Spring AOP – for aspect-oriented programming

>> Spring DAO – for database operation using DAO pattern

>> Spring JDBC – for JDBC & DataSource support

>> Spring ORM – for ORM tools support such as Hibernate

>> Spring Web Model – for creating web application

>> Spring MVC (Model view controller implementation) – for creating web application, web service etc

\_**List some of the important annotations in annotation-based Spring configuration**\_

>>

@Required: applies to bean property setter methods.

@Autowired: apply to bean property setter methods, non-setter methods, constructor

and properties.

@Qualifier: annotation along with @Autowired can be used to remove the

confusion by specifiying which exact bean will be wired.

@PostConstruct: annotation defines a method as initialization method of a spring bean

which runs after dependency injection iscompleted. The method is used to validate properties of bean or initializing any task.

@PreDestroy: annotation declares a method that is executed just before the bean is

destroyed by container. The method used to release the resources or preforming any destruction task before the spring container destroyed the bean.

@Resource: It behaves same as @Autowired annotation of spring. @Autowired annotation automatically injects the

required bean. @Resource has name attribute.

\_\_\_ **What is a Bean in Spring & explain the different scopes of bean in Spring?** \_\_

>> A Bean is an object that instantiated, assembled, and managed by a Spring loC container.

>> They are maaged by the Spring loC container.

>> Ther are 5 Scopes of Beans in Spring:

|  |
| --- |
| **Global-Session**  **Scope**  **Prototype**  **Request**  **Session**  **Singleton** |

>> In the spring bean configurations, bean attribute called 'scope' defines what kind of object has to created and returned. There are 5 types of bean scopes available, they are:

**1) singleton:**Returns a single bean instance per Spring IoC container.

**2) prototype:**Returns a new bean instance each time when requested.

**3) request:**Returns a single instance for every HTTP request call.

**4) session:**Returns a single instance for every HTTP session.

**5) global session:**global session scope is equal as session scope on portlet-based web applications.

If no bean scope is specified in bean configuration file, then it will be by default 'singleton'.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ **How is a Bean added to a Spring application?** \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

<?xml version =”1.0 encoding+”UTF-8”?>

<!DOCTYPE beans PUBLIC “ -//spring//ATD BEAN//EN”>

<beans>

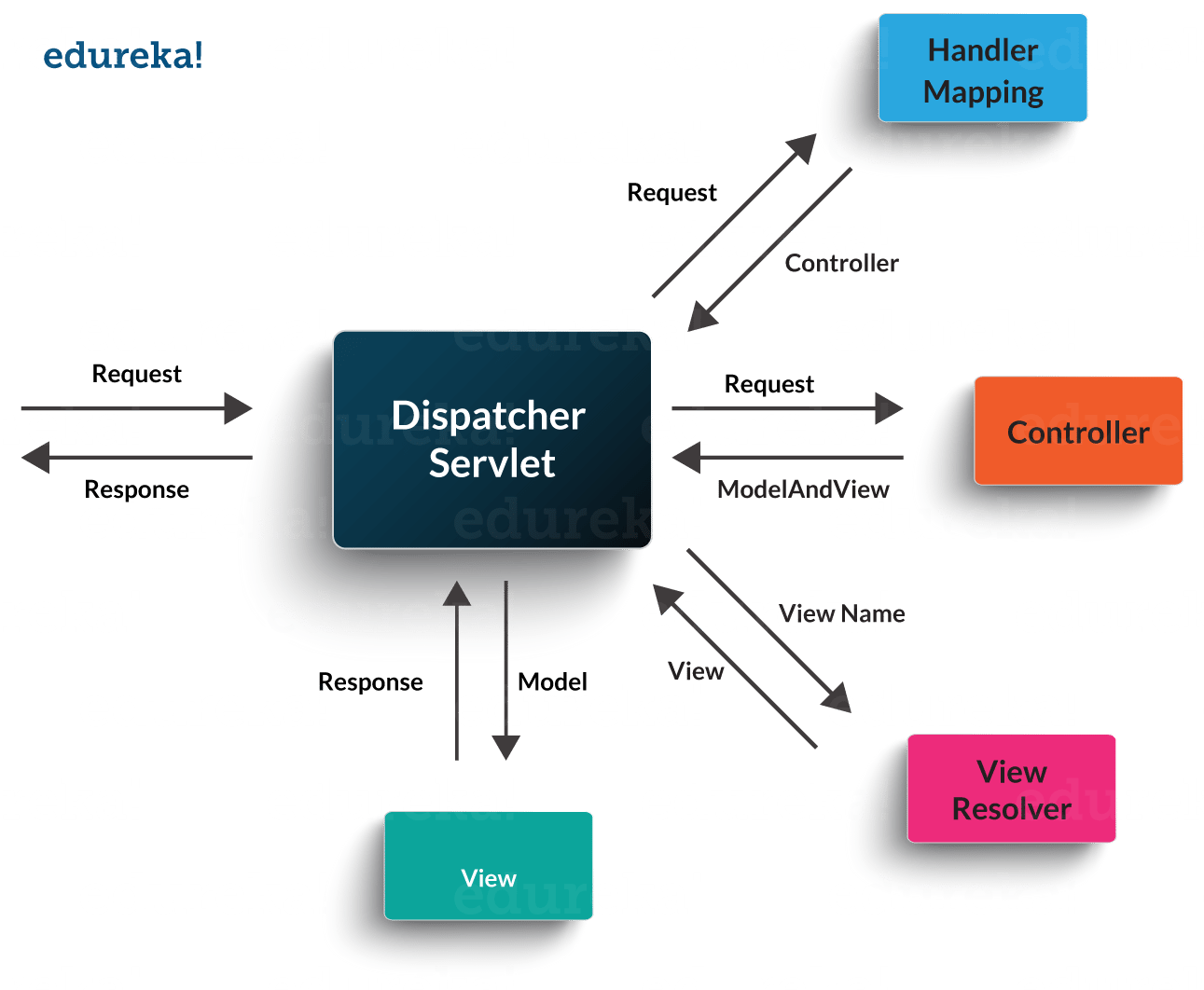
<bean id = “foo”/>

<bean id = “bar”/>

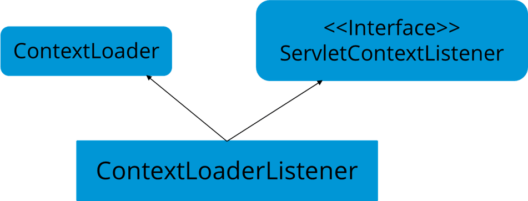
</beans>

\_\_\_\_\_\_\_\_\_\_\_ **Explain the role of Dispatcher & ContextLoaderListener?** \_\_\_\_\_\_\_\_\_\_\_\_

>> Dispatcher Servlet: is the ront controller in the SpringMVC application as it loads the spring bean configuration file & initializes all the beans that have bean configured.



>> ContextLoaderListener: is the listener to start up & shut down the WebApplicationContext in Spring root.



\_\_ **What are the differences between constructor injection and setter injection?** \_

|  |  |  |
| --- | --- | --- |
| **No.** | **Constructor Injection** | **Setter Injection** |
| 1) | No Partial Injection | Partial Injection |
| 2) | Desn’t override the setter property | Overrides the constructor property if both are defined. |
| 3) | Creates new instance if any modification occurs | Doesn’t create new instance if you change the property value |
| 4) | Better for too many properties | Better for few properties. |

\_\_\_\_\_\_\_\_\_ **What is autowiring in Spring? What are the autowiring modes?** \_\_\_\_\_\_\_\_\_\_

>> Autowiring enables the programmer to inject the bean automatically. We don’t need to write explicit injection logic. Let’s see the code to inject bean using dependency injection.

1. <bean id=“emp” class=“com.javatpoint.Employee” autowire=“byName” />

The autowiring modes are given below:

|  |  |  |
| --- | --- | --- |
| **No.** | **Mode** | **Description** |
| 1) | no | this is the default mode, it means autowiring is not enabled. |
| 2) | byName | Injects the bean based on the property name. It uses setter method. |
| 3) | byType | Injects the bean based on the property type. It uses setter method. |
| 4) | constructor | It injects the bean using constructor |

\_\_\_\_\_\_\_\_\_\_\_\_\_ **How to handle exceptions in Spring MVC Framework?** \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**>> Spring MVC Framework provides the following ways to help us achieving robust exception handling.**

**Controller Based:** We can define exception handler methods in our controller classes. All we need is to annotate these methods with @ExceptionHandler annotation.

**Global Exception Handler:** Exception Handling is a cross-cutting concern and Spring provides @ControllerAdvice annotation that we can use with any class to define our global exception handler.

**HandlerExceptionResolver implementation:** For generic exceptions, most of the times we serve static pages. Spring Framework provides HandlerExceptionResolver interface that we can implement to create global exception handler. The reason behind this additional way to define global exception handler is that Spring framework also provides default implementation classes that we can define in our spring bean configuration file to get spring framework exception handling benefits.

### **What are some of the important Spring annotations which you have used?**

Some of the Spring annotations that I have used in my project are:

**@Controller** – for controller classes in Spring MVC project.

**@RequestMapping** – for configuring URI mapping in controller handler methods. This is a very important annotation, so you should go through Spring MVC RequestMapping Annotation Examples

**@ResponseBody** – for sending Object as response, usually for sending XML or JSON data as response.

**@PathVariable** – for mapping dynamic values from the URI to handler method arguments.

**@Autowired** – for autowiring dependencies in spring beans.

**@Qualifier** – with @Autowired annotation to avoid confusion when multiple instances of bean type is present.

**@Service** – for service classes.

**@Scope** – for configuring scope of the spring bean.

**@Configuration, @ComponentScan and @Bean** – for java based configurations.

AspectJ annotations for configuring aspects and advices, @Aspect, @Before, @After, @Around, @Pointcut etc.

### **How to integrate Spring and Hibernate Frameworks?**

We can use Spring ORM module to integrate Spring and Hibernate frameworks.

Spring ORM provides support for using Spring declarative transaction management.

## **Hibernate:**

### **What is Hibernate Framework?**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_­­­­\_\_\_ **Three types of Iteration** \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

|  |
| --- |
| **1, Enumeration:** It is an[**interface**](https://data-flair.training/blogs/interface-in-java/) used to get components of [**inheritance**](https://data-flair.training/blogs/inheritance-in-java/) collections (Vector, Hashtable).   * It is for a legacy class, and hence not a universal Java iterator. * The operations of removing cannot be performed. * Only forward iteration is possible.   **Syntax –**  Enumeration e = v.elements();    //The two methods in enumeration **interface** are –    // Tests if this enumeration contains more elements  **public** **boolean** hasMoreElements();  // Returns the next element of this enumeration  // It throws NoSuchElementException  // if no more element present  **public** Object nextElement();  **Example of Enumeration-**  Vector v = **new** Vector();    Enumeration e = v.elements();    **while** (e.hasMoreElements()) {  **int** i = (Integer) e.nextElement();  System.***out***.print(i + " "); **2, Iterator:** Java iterator is the main cursor accessible for whole collection Framework.  It is a general (universal) Java iterator as we can apply it to any collection object. By utilizing Java Iterator, we can perform both read and remove operations.  structure actualized interfaces like Set, List, Queue, Deque and furthermore in every single executed class of Map interface.  **A Syntax of Java Iterator –**  Iterator itr = c.iterator();  // Returns true if the iteration has more elements  **public** **boolean** hasNext();  // Returns the next element in the iteration  // It throws NoSuchElementException if no more  // element present  **public** Object next();  // Remove the next element in the iteration  // This method can be called only once per call  // to next()  **public** **void** remove();  **Example of Iterator –**  ArrayList al = **new** ArrayList();  Iterator itr = al.iterator();  **while** (itr.hasNext())  {  **int** i = (Integer)itr.next();  System.out.print(i + " ");  } **3, ListIterator:** This is only applicable where there is List implemented classes such as ArrayList, linked list etc. It provides the user to give bi-directional iteration, it is used when we want to enumerate the list, it has more functionality.  ArrayList al = **new** ArrayList();  ListIterator ltr = al.listIterator();  **while** (ltr.hasNext())  {  **int** i = (Integer)ltr.next();  System.out.print(i + " ");  }  1) void add(E e): Inserts the specified element into the list (optional operation).  2) boolean hasNext(): Returns true if this list iterator has more elements when  traversing the list in the forward direction.  3) boolean hasPrevious(): Returns true if this list iterator has more elements when  traversing the list in the reverse direction.  4) E next(): Returns the next element in the list and advances the cursor position.  5) int nextIndex(): Returns the index of the element that would be returned by a  subsequent call to next().  6) E previous(): Returns the previous element in the list and moves the cursor position  backwards.  7) int previousIndex(): Returns the index of the element that would be returned by a  subsequent call to previous().  8) void remove(): Removes from the list the last element that was returned by next() or  previous() (optional operation).  9) void set(E e): Replaces the last element returned by next() or previous() with the  specified element (optional operation). |

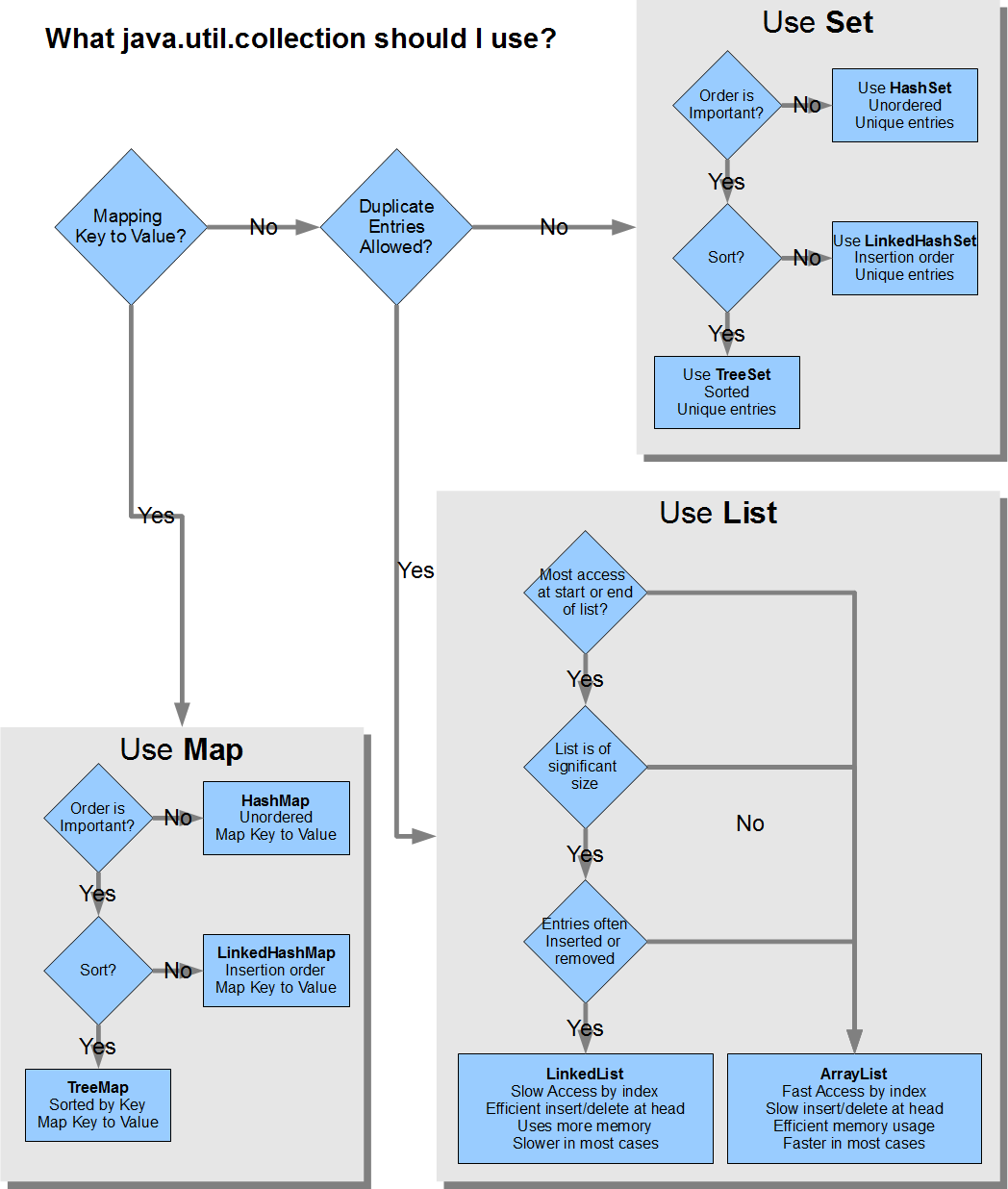
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_­­­­\_\_\_\_\_\_\_\_\_ **Java Collection** \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

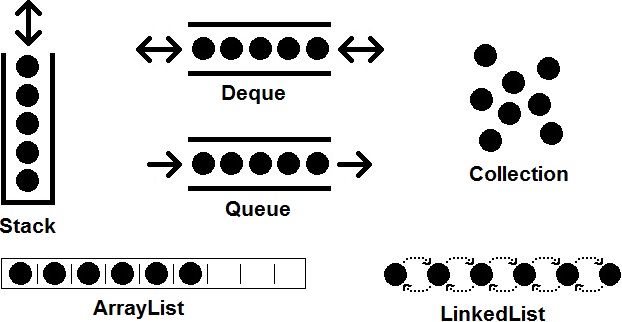
>> The **Collection in Java** is a framework that provides an architecture to store and manipulate the

group of objects.

**Hirarchy of java Colletcion:**

|  |
| --- |
| Hierarchy of Java Collection framework |

****

****

**Methods of Collection:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **No.** | | **Method** | **Description** | |
| 1 | public boolean add(E e) | | | It is used to insert an element in this collection. |
| 2 | public boolean addAll(Collection<? extends E> c) | | | It is used to insert the specified collection elements in the invoking collection. |
| 3 | public boolean remove(Object element) | | | It is used to delete an element from the collection. |
| 4 | public boolean removeAll(Collection<?> c) | | | It is used to delete all the elements of the specified collection from the invoking collection. |
| 5 | default boolean removeIf(Predicate<? super E> filter) | | | It is used to delete all the elements of the collection that satisfy the specified predicate. |
| 6 | public boolean retainAll(Collection<?> c) | | | It is used to delete all the elements of invoking collection except the specified collection. |
| 7 | public int size() | | | It returns the total number of elements in the collection. |
| 8 | public void clear() | | | It removes the total number of elements from the collection. |
| 9 | public boolean contains(Object element) | | | It is used to search an element. |
| 10 | public boolean containsAll(Collection<?> c) | | | It is used to search the specified collection in the collection. |
| 11 | public Iterator iterator() | | | It returns an iterator. |
| 12 | public Object[] toArray() | | | It converts collection into array. |
| 13 | public <T> T[] toArray(T[] a) | | | It converts collection into array. Here, the runtime type of the returned array is that of the specified array. |
| 14 | public boolean isEmpty() | | | It checks if collection is empty. |
| 15 | default Stream<E> parallelStream() | | | It returns a possibly parallel Stream with the collection as its source. |
| 16 | default Stream<E> stream() | | | It returns a sequential Stream with the collection as its source. |
| 17 | default Spliterator<E> spliterator() | | | It generates a Spliterator over the specified elements in the collection. |
| 18 | public boolean equals(Object element) | | | It matches two collections. |
| 19 | public int hashCode() | | | It returns the hash code number of the collection. |

## **List Interface:**

List interface is the child interface of Collection interface. It inhibits a list type data structure in which we can store the ordered collection of objects. It can have duplicate values.

List interface is implemented by the classes **ArrayList, LinkedList, Vector, and Stack.**

To instantiate the List interface, we must use :

**1, List <data-type> list1= new ArrayList();**

**2, List <data-type> list2 = new LinkedList();**

**3, List <data-type> list3 = new Vector();**

**4, List <data-type> list4 = new Stack();**

## **ArrayList:**

It uses a dynamic array to store the duplicate element of different data types.

* **Java ArrayList class can contain duplicate elements.**
* **Java ArrayList class maintains insertion order.**
* **Java ArrayList class is non synchronized.**
* **Java ArrayList allows random access because array works at the index basis.**
* **In Java ArrayList class, manipulation is slow because a lot of shifting needs to occur if any element is removed from the array list.**

There are various ways to traverse the collection elements**:**

**By Iterator interface, 2. By for-each loop, 3. By ListIterator interface, 4. By for loop, 5. By forEach() method, 6. By forEachRemaining() method.**

|  |
| --- |
| **import** java.util.\*;  **class** InterviewQuestion {  **public** **static** **void** main(String[] args) {  ArrayList<String> list = **new** ArrayList<String>();// Creating arraylist  list.add("Ravi");// Adding object in arraylist  list.add("Vijay");  list.add("Ravi");  list.add("Ajay");  // Traversing list through Iterator  Iterator itr = list.iterator();  **while** (itr.hasNext()) {  System.***out***.println(itr.next());  }  // Traversing list through List Iterator  ListIterator<String> list1 = list.listIterator(list.size());  **while** (list1.hasPrevious()) {  String str = list1.previous();  System.***out***.println(str);  }  // Traversing list through for loop  **for** (**int** i = 0; i < list.size(); i++) {  System.***out***.println(list.get(i));  }  // Traversing list through forEach() method:  // The forEach() method is a new feature, introduced in Java 8.  list.forEach(a -> { // Here, we are using lambda expression  System.***out***.println(a);  });  // Traversing list through forEachRemaining() method:  Iterator<String> it = list.iterator();  it.forEachRemaining(a -> // Here, we are using lambda expression  {  System.***out***.println(a);  });  }  } |

## **LinkedList:**

LinkedList implements the Collection interface. It uses a doubly linked list internally to store the elements. It can store the duplicate elements. It maintains the insertion order and is not synchronized. In LinkedList, the manipulation is fast because no shifting is required.

|  |
| --- |
| **import** java.util.\*;  **class** InterviewQuestion {  **public** **static** **void** main(String[] args) {  LinkedList<String> al = **new** LinkedList<String>(); // Create LinkedList  al.add("Ravi");  al.add("Vijay");  al.add("Ravi");  al.add("Ajay");    Iterator<String> itr = al.iterator(); // Traversing list through Iterator  **while** (itr.hasNext()) {  System.***out***.println(itr.next());  }  }  } |

Consider the following example.

## **Vector:**

## Vector uses a dynamic array to store the data elements. It is similar to ArrayList. However, It is synchronized and contains many methods that are not the part of Collection framework.

Consider the following example.

|  |
| --- |
| **import** java.util.\*;  **class** InterviewQuestion {  **public** **static** **void** main(String[] args) {  Vector<String> v = **new** Vector<String>();  v.add("Ayush");  v.add("Amit");  v.add("Ashish");  v.add("Garima");  Iterator<String> itr = v.iterator();  **while** (itr.hasNext()) {  System.***out***.println(itr.next());  }  }  } |

## **Stack:**

The stack is the subclass of Vector. It implements the last-in-first-out data structure, i.e., Stack. The stack contains all of the methods of Vector class and also provides its methods like boolean push(), boolean peek(), boolean push(object o), which defines its properties.

Consider the following example.

|  |
| --- |
| **import** java.util.\*;  **class** InterviewQuestion {  **public** **static** **void** main(String[] args) {  Stack<String> stack = **new** Stack<String>();  stack.push("Ayush");  stack.push("Garvit");  stack.push("Amit");  stack.push("Ashish");  stack.push("Garima");  stack.pop();  Iterator<String> itr = stack.iterator();  **while** (itr.hasNext()) {  System.***out***.println(itr.next());  }  }  } |

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ **Difference between ArrayList & Vector** \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

|  |  |
| --- | --- |
| **ArrayList** | **Vector** |
| Array List is not synchronized | vector is synchronized |
| Array List is fast as it's not synchronized | vector is slow as it is thread safe |
| If an element is inserted into the Array List, it increases it's Array size by 50%. | vector defaults to doubling size of its Array. |
| Array List doesn't define the incremenet size | Vector defines the increment size. |
| Array list can only use iterator for traversing an Array list | Except Hashtable, Vector is the only other class which uses both enumeration & iteration |

## **Queue Interface:**

Queue interface maintains the first-in-first-out order. It can be defined as an ordered list that is used to hold the elements which are about to be processed. There are various classes like PriorityQueue, Deque, and ArrayDeque which implements the Queue interface.

Queue interface can be instantiated as:

1, Queue<String> q1 = **new** PriorityQueue();

2, Queue<String> q2 = **new** ArrayDeque();

There are various classes that implement the Queue interface, some of them are given below.

## **PriorityQueue:**

The PriorityQueue class implements the Queue interface. It holds the elements or objects which are to be processed by their priorities. PriorityQueue doesn't allow null values to be stored in the queue.

Consider the following example.

|  |
| --- |
| **import** java.util.\*;  **class** InterviewQuestion {  **public** **static** **void** main(String[] args) {  PriorityQueue<String> queue = **new** PriorityQueue<String>();  queue.add("Amit Sharma");  queue.add("Vijay Raj");  queue.add("JaiShankar");  queue.add("Raj");    System.***out***.println("head:" + queue.element());  System.***out***.println("head:" + queue.peek());  System.***out***.println("iterating the queue elements:");    Iterator itr = queue.iterator();  **while** (itr.hasNext()) {  System.***out***.println(itr.next());  }  queue.remove();  queue.poll(); // identifies the head and then removes it.  System.***out***.println("after removing two elements:");  Iterator<String> itr2 = queue.iterator();  **while** (itr2.hasNext()) {  System.***out***.println(itr2.next());  }  }  }  >> Out put  head:Amit Sharma  head:Amit Sharma  iterating the queue elements:  Amit Sharma  Raj  JaiShankar  Vijay Raj  after removing two elements:  Raj  Vijay Raj |

## **Deque Interface:**

Deque interface extends the Queue interface. In Deque, we can remove and add the elements from both the side. Deque stands for a double-ended queue which enables us to perform the operations at both the ends.

Deque can be instantiated as:

1, **Deque d = new ArrayDeque();**

## **ArrayDeque:**

ArrayDeque class implements the Deque interface. It facilitates us to use the Deque. Unlike queue, we can add or delete the elements from both the ends.

ArrayDeque is faster than ArrayList and Stack and has no capacity restrictions.

Consider the following example.

|  |
| --- |
| **import** java.util.\*;  **class** InterviewQuestion {  **public** **static** **void** main(String[] args) {  // Creating Deque and adding elements  ArrayDeque<String> deque = **new** ArrayDeque<String>();  deque.add("Gautam");  deque.add("Karan");  deque.add("Ajay");    // Traversing elements  **for** (String str : deque) {  System.***out***.println(str);  }  }  } |

## **Set Interface:**

Set Interface in Java is present in java.util package. It extends the Collection interface. It represents the unordered set of elements which doesn't allow us to store the duplicate items. We can store at most one null value in Set. Set is implemented by HashSet, LinkedHashSet, and TreeSet.

Set can be instantiated as:

1, Set<data-type> s1 = **new** HashSet<data-type>();

2, Set<data-type> s2 = **new** LinkedHashSet<data-type>();

3, Set<data-type> s3 = **new** TreeSet<data-type>();

## **HashSet:**

HashSet class implements Set Interface. It represents the collection that uses a hash table for storage. Hashing is used to store the elements in the HashSet. It contains unique items.

|  |
| --- |
| **import** java.util.\*;  **class** InterviewQuestion {  **public** **static** **void** main(String[] args) {  // Creating HashSet and adding elements  HashSet<String> set = **new** HashSet<String>();  set.add("Ravi");  set.add("Vijay");  set.add("Ravi");  set.add("Ajay");    // Traversing elements  Iterator<String> itr = set.iterator();  **while** (itr.hasNext()) {  System.***out***.println(itr.next());  }  }  } |

Consider the following example.

## **LinkedHashSet:**

## LinkedHashSet class represents the LinkedList implementation of Set Interface. It extends the HashSet class and implements Set interface. Like HashSet, It also contains unique elements. It maintains the insertion order and permits null elements.

|  |
| --- |
| **import** java.util.\*;  **class** InterviewQuestion {  **public** **static** **void** main(String[] args) {  LinkedHashSet<String> set = **new** LinkedHashSet<String>();  set.add("Ravi");  set.add("Vijay");  set.add("Ravi");  set.add("Ajay");  Iterator<String> itr = set.iterator();  **while** (itr.hasNext()) {  System.***out***.println(itr.next());  }  }  } |

Consider the following example.

## **SortedSet Interface:**

SortedSet is the alternate of Set interface that provides a total ordering on its elements. The elements of the SortedSet are arranged in the increasing (ascending) order. The SortedSet provides the additional methods that inhibit the natural ordering of the elements.

The SortedSet can be instantiated as:

1, SortedSet<data-type> set = **new** TreeSet();

## **TreeSet:**

Java TreeSet class implements the Set interface that uses a tree for storage. Like HashSet, TreeSet also contains unique elements. However, the access and retrieval time of TreeSet is quite fast. The elements in TreeSet stored in ascending order.

Consider the following example:

|  |
| --- |
| **import** java.util.\*;  **class** InterviewQuestion {  **public** **static** **void** main(String[] args) {  // Creating and adding elements  TreeSet<String> set = **new** TreeSet<String>();  set.add("Ravi");  set.add("Vijay");  set.add("Ravi");  set.add("Ajay");    // traversing elements  Iterator<String> itr = set.iterator();  **while** (itr.hasNext()) {  System.***out***.println(itr.next());  }  }  } |

**Google Sample Interview Question In Collection:**

SDE candidates need to demonstrate a basic understanding of the most common data structures, and of the fundamentals of "big-O" algorithmic complexity analysis.

Here's what they need to know about big-O. They need to know that algorithms usually fall into the following performance classes: constant-time, logarithmic, linear, polynomial, exponential, and factorial.

For the standard data structures in java.util, STL, or those built into a higher-level language, they need to know the big-O complexity for the operations on those data structures.  *Example:* they should know that finding an element in a hashtable is usually constant-time, that finding an element in a balanced binary tree is order log(n), that finding an element in a linked list is order N, and that finding an element in a sorted array is order log(n). Similarly for insert/update/delete operations.

And they should be able to explain why each operation falls into a particular complexity class. For instance: "Computing a hash value doesn't depend on the number of items in the hashtable." Or: "you have to search the entire linked list, even if it's sorted, to find an arbitrary element in it." No math needed, no proofs, just explanations.

The (concrete) data structures they absolutely must understand are these:

1) *arrays* - I'm talking about C-language and Java-language arrays: fixed-sized, indexed, contiguous structures whose elements are all of the same type, and whose elements can be accessed in constant time given their indices.

2) *vectors* - also known as "growable arrays" or ArrayLists. Need to know that they're objects that are backed by a fixed-size array, and that they resize themselves as necessary.

3) *linked lists* - lists made of nodes that contain a data item and a pointer/reference to the next (and possibly previous) node.

4) *hashtables* - amortized constant-time access data structures that map keys to values, and are backed by a real array in memory, with some form of collision handling for values that hash to the same location.

5) *trees* - data structures that consist of nodes with optional data elements and one or more child pointers/references, and possibly parent pointers, representing a heirarchical or ordered set of data elements.

6) *graphs* - data structures that represent arbitrary relationships between members of any data set, represented as networks of nodes and edges.

There are, to be sure, many other important data structures one should know about, but not knowing about the six listed above is inexcusable, and grounds for rejection in a phone screen.

Candidates should be able to describe, for any of the data structures above:

* what you use them for (real-life examples)
* why you prefer them for those examples
* the operations they typically provide (e.g. insert, delete, find)
* the big-O performance of those operations (e.g. logarithmic, exponential)
* how you traverse them to visit all their elements, and what order they're visited in
* at least one typical implementation for the data structure

Candidates should know the difference between an abstract data type such as a Stack, Map, List or Set, and a concrete data structure such as a singly-linked list or a hash table. For a given abstract data type (e.g. a Queue), they should be able to suggest at least two possible concrete implementations, and explain the performance trade-offs between the two implementations.

Example weeder questions:

1) *What are some really common data structures, e.g. in java.util?*

2) *When would you use a linked list vs. a vector?*

3) *Can you implement a Map with a tree? What about with a list?*

4) *How do you print out the nodes of a tree in level-order (i.e. first level, then 2nd level, then 3rd level, etc.)*

5) *What's the worst-case insertion performance of a hashtable? Of a binary tree?*

6) *What are some options for implementing a priority queue?*

And so on. Just a few quick questions should cover this area, provided you don't focus exclusively on linear ordered sequences (lists, arrays, vectors and the like).

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ **Java Map** \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Java Maps Interface:**

1. An object that maps keys to values.
2. A map cannot contain duplicate keys
3. each key can map to at most one value.
4. This interface takes the place of the Dictionary class, which was a totally

abstract class rather than an interface.

1. Map doesn’t guarantee order of mappings, however it depends on the

implementation. For example, HashMap doesn’t guarantee the order of

mappings but TreeMap does.

1. Map utilize hashCode and equals methods on Key for get and put operations.

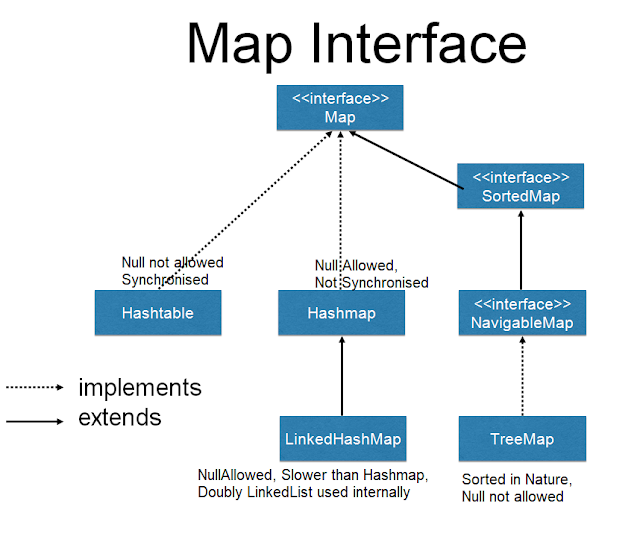
So mutable classes are not a good fit for Map keys. If the values of hashCode

or equals change after put, you won’t get the correct value in get operation

**Why and when to use maps?**

1. A map of error codes and their descriptions.
2. A map of zip codes and cities.
3. A map of employees and employee ID.
4. A map of classes and students.

**Hirarchy of Map:**

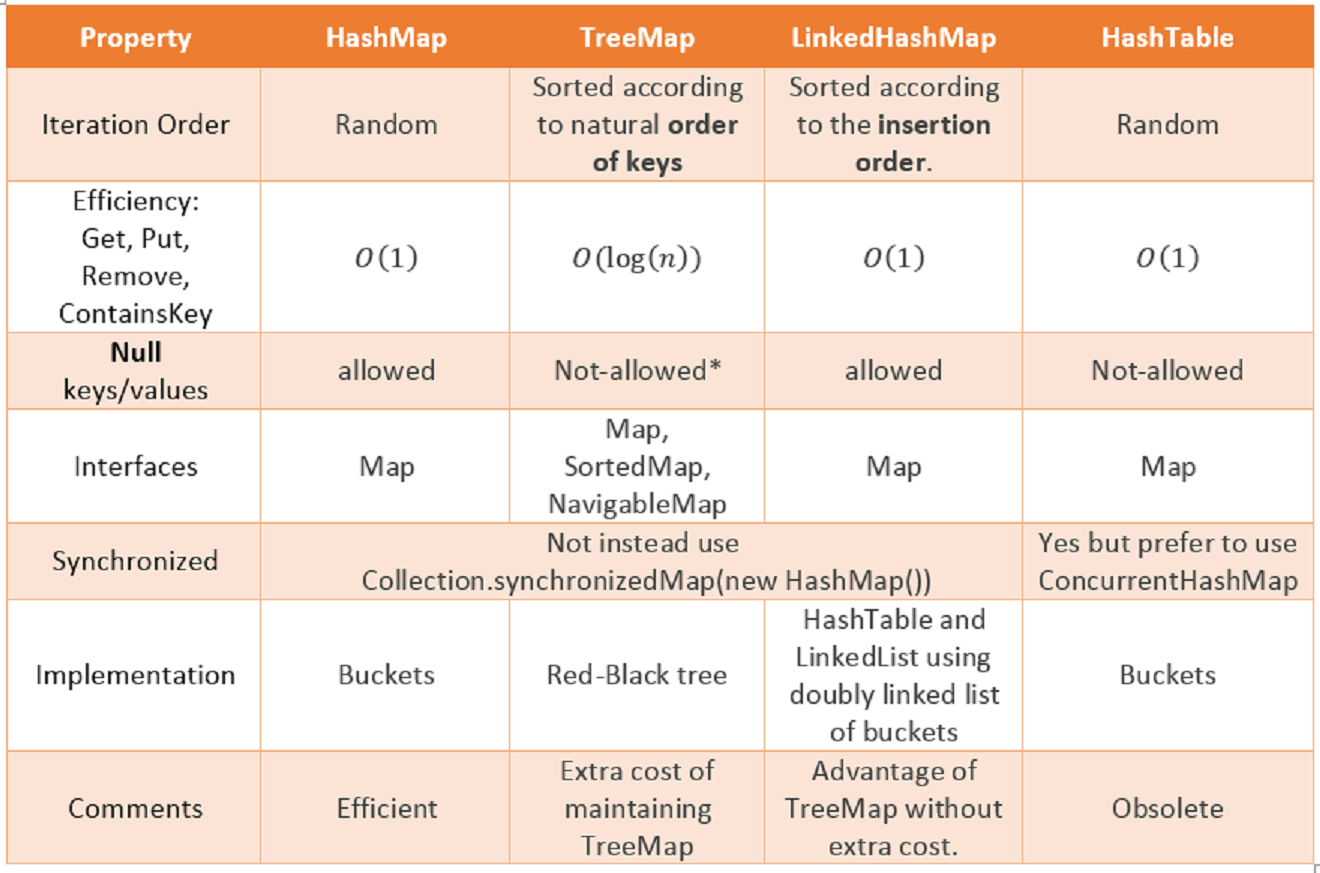


**Four types of Map:**

|  |
| --- |
| **HashMap and Hashtable**doesn’t maintain any order.  **TreeMap**sort the entries in ascending order of keys.  **LinkedHashMap**maintains the insertion order.  **HashMap:** 1, It is similar to the Hashtable class except that it is unsynchronized and permits null.  2, It may have one null key and multiple null values.  3, It is fast & fail-fast.  4, It is traversed by Iterator & sorted in the ascending order of its keys  **HashTable: 1,** Any non-null object can be used as a key or as a value.  2, It may have not have any null key or value.  3, It is synchronized an thread-safe.  4, Traversed by Enumerator and sorted in the descending order of its keys  5, It is not fail-fast, it is slow.  **LinkedHashMap:** 1, Contains values based on the key.  2, It may have one null key and multiple null values.  3, This linked list defines the iteration in the order of entering key & value.  **TreeMap:** 1, It is sorted in the ascending order of its keys  2, It is unsynchronized collection class. |

**Differences :**

****



**All usefull methods:**

|  |
| --- |
| **1) Object put(Object key, Object value) :**is used to insert an entry in this map.  **2) void putAll(Map map) :**is used to insert the specified map in this map.  **3) Object remove(Object key) :**is used to delete an entry for the specified key.  **4) Object get(Object key) :**is used to return the value for the specified key.  **5) boolean containsKey(Object key) :**is used to search the specified key from this map.  **6) boolean containsValue(Object value):**returns true if there are at least one key mapped to the specified  value, otherwise false.  **7) Set keySet() :**returns the Set view containing all the keys.  **8) Set entrySet() :**returns the Set view containing all the keys and values.  **9) int size():**returns the number of key-value mappings in this Map.  **10) boolean isEmpty():**returns true if there are no mappings present, otherwise false.  **11) void clear():**removes all the mappings from the Map.  **10) Collection<V> values():** returns the collection view of all the values in the Map. This collection is  backed by Map, so any change in Map will reflect to this values collection and vice versa.  **11) Set<Map.Entry<K, V>> entrySet():** returns the Set view of the mappings in the Map. This Set is  backed by Map, so any modifications in Map will be reflected in the  entry set and vice versa. Maps are further classified into the following: |

## **Map.Entry Interface:**

## Entry is the sub interface of Map. So we will be accessed it by Map.Entry name. It provides methods to get key and value.

|  |  |
| --- | --- |
| **Method** | **Description** |
| Object getKey() | It is used to obtain key. |
| Object getValue() | It is used to obtain value. |

|  |
| --- |
| **Generic new Style**  Map<Integer,String> map = **new** HashMap<Integer,String>();    **for**(Map.Entry m : map.entrySet())  {  System.out.println(m.getKey()+" "+m.getValue());  }  **Non-generic old Style:**  Set set = map.entrySet(); //Converting to Set so that we can traverse    Iterator itr = set.iterator();    **while**(itr.hasNext())  {  //Converting to Map.Entry so that we can get key and value separately  Map.Entry entry = (Map.Entry)itr.next();  System.out.println(entry.getKey()+" "+entry.getValue());  } |

**Example of Map Iteration:**

A simple way to see the key value pairs:

Map<String, Integer> map = new HashMap<>();

map.put("a", 1);

map.put("b", 2);

System.out.println(Arrays.asList(map)); // method 1

System.out.println(Collections.singletonList(map)); // method 2

Both method 1 and method 2 output this:

[{b=2, a=1}]

Map<String, Object> map = ...;

for (String key : map.keySet()) {

// ...

}

If you only need the values, use [values()](http://java.sun.com/j2se/1.5.0/docs/api/java/util/Map.html#values%28%29):

for (Object value : map.values()) {

// ...

}

Finally, if you want both the key and value, use [entrySet()](http://java.sun.com/j2se/1.5.0/docs/api/java/util/Map.html" \l "entrySet%28%29):

for (Map.Entry<String, Object> entry : map.entrySet()) {

String key = entry.getKey();

Object value = entry.getValue();

// ...

}

**or**

for (Object obj : dados.entrySet()) {

Map.Entry<String, String> entry = (Map.Entry) obj;

System.out.print("Key: " + entry.getKey());

System.out.println(", Value: " + entry.getValue());

}

**or**

for (Object objectName : example.keySet()) {

System.out.println(objectName);

System.out.println(example.get(objectName));

}

|  |
| --- |
| **import** java.util.\*;  **class** InterviewQuestion {  **public** **static** **void** main(String[] args) {  Hashtable<Integer, String> hm = **new** Hashtable<Integer, String>();  hm.put(100, "Amit");  hm.put(102, "Ravi");  hm.put(101, "Vijay");  hm.put(103, "Rahul");  // Iterating the hash table  **for** (Map.Entry m : hm.entrySet()) {  System.***out***.println(m.getKey() + " " + m.getValue());  }  }  }  >> Output 103 Rahul  102 Ravi  101 Vijay  100 Amit |

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**Difference between HashMap & HashTable**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

|  |  |
| --- | --- |
| **HashMap** | **HahsTable** |
| HashMap is non synchronized and are not thread safe | HashTable is synchronized and it is thread safe and can be shared with many threads |
| HashMap allows one null key and multiple null values | HashTable doesn't allow any null key or value. |
| Hasmap is fast comparing to HashTable | HashTable is slow compared to HashMap |
| We can make HashMap as synchronized by calling collections.synchronizedMap(hasMap) | HashTable is internally synchronized and can't be unsynchronized. |
| HashMap is traversed by iteration | Hashtable is traversed by Enumerator and iterator. |
| HashMap inherits AbstractMap class | HashTable is inherits Dictionary class |

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ **Analysis of Algorithm Complexity** \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Asymptotic Notations**

**Asymptotic Notations** are languages that allow us to analyze an algorithm’s running time by identifying its behavior as the input size for the algorithm increases. This is also known as an algorithm’s growth rate.

The following 3 asymptotic notations are mostly used to represent time complexity of algorithms:

**Big Oh (O)**

Big Oh is often used to describe the worst-case of an algorithm by taking the highest order of a polynomial function and ignoring all the constants value since they aren’t too influential for sufficiently large input.

**Big Omega (Ω)**

Big Omega is the opposite of Big Oh, if Big Oh was used to describe the upper bound (worst-case) of a asymptotic function, Big Omega is used to describe the lower bound of a asymptotic function. In analysis algorithm, this notation is usually used to describe the complexity of an algorithm in the best-case, which means the algorithm will not be better than its **best-case**.

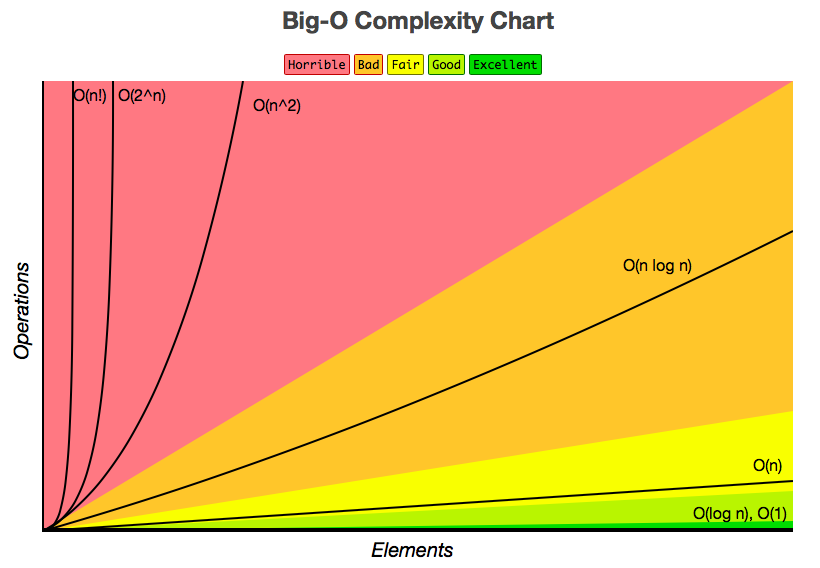
**Big Theta (Θ)**

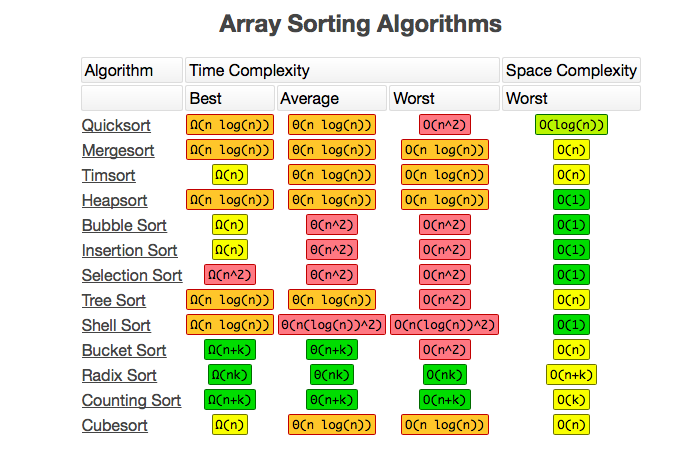
When an algorithm has a complexity with lower bound = upper bound, say that an algorithm has a complexity O(n log n) and Ω(n log n), it’s actually has the complexity Θ(n log n), which means the running time of that algorithm always falls in n log n in the best-case and worst-case.

**Algorithmic Examples of Runtime Analysis**:



**Where, n is the input size and c is a positive constant.**



****

Determining Time Complexity:

**x = 0;  
 for (int i = 0; i < n; i++)**

**x = x + 1;**

In each iteration of the loop four operations are performed **(<, ++, =, +)**. Since the loop is repeated n times, the total number of operations performed is **4n+1**

Notice any constant factors in the time complexity function are ignored when computing the order of the time complexity. Therefore, time complexity of (**4n+1) is O(n).**

So, that the time complexity of an algorithm is the same as the number of primitive operations that it performs.

However, we need to be very careful when counting the number of iterations of any loop.

When computing the time complexity of an algorithm with nested loops, like the algorithm below, we usually consider the innermost loop first and work our way outward.

**for (int i=0; i<n; i++)**

**{**

**x = x + 1;**

**for (int j=0; j<n; j++)**

**y = y – 1;**

**}**

Each iteration of the inner loop performs a constant number k of operations. Every time that the loop is performed, it iterates n times (as j takes values 0, 1, ..., n-1), therefore the total number of operations performed by the inner loop is kn.

Outside the inner loop, but inside the outer one, an additional constant number k’ of operations are performed **(x = x+1),** so each iteration of the outside loop performs **k’ + kn** operations.

The outer loop is repeated n times, so the total number of operations performed by this algorithm is **t(n) = n(k’ + kn) = k’n + kn2.** Note that in this function the term **kn2** asymptotically dominates the term **k’n,** so ignoring constant factor we get that **t(n) is O(n2).**

**Another example:**

**x = 0;  
for (int i=0; i<n; i=i+2)**

**{**

**x = x + 1;**

**}**

For the above algorithm, each iteration of the loop performs a constant number k of operations. The loop is repeated **n/2 times**, as in each iteration the value of i increases by **2**. Furthermore, outside the loop an additional constant number **k’** of operations are performed.

The total number of operations performed by the algorithm is **k’ + kn/2.** The time complexity of this algorithm, then is **O(n).**

**More Loop Analysis Examples:**

**x = 0;  
for (int i=1; i<=n; i=i\*2) {**

**x = x + 1; }**

For the above algorithm, outside the loop a constant number **k’** of operations are performed. Also, each iteration of the loop performs a constant number **k** of operations. Counting the total number of iterations of the loop is more complicated. Let us look at the value of **i** in each iteration:

|  |  |
| --- | --- |
| **Iteration number** | **Value of i** |
| 1 | 1 = 20 |
| 2 | 2 = 21 |
| 3 | 4 = 22 |
| 4 | 8 = 23 |
| ... | ... |
| **1 + log n** | **n = 2log n** |

Hence, the number of iterations performed by the loop is **1 + log n**. The total number of operations performed by the loop is **t(n) = k’ + k(1 + log n) = k’ + k + k log n**. The dominating term is **k log n**, so the time complexity is **O(log n)**.

**More Loop Analysis Examples:**

**x = 0;  
for (int i=0; i<n; i++)**

**for (int j = i, j < n, j ++)**

**{**

**x = x + 1;**

**}**

The inner loop performs a constant number k of operations in each iteration. The inner loop repeats once for each value of j between i and n-1, thus the number of iterations is n-i. The number of operations performed by the inner loop is then k(n-i). Note that this expression depends on the value of the variable i.

The outer loop is repeated once for each value of i between 0 and n-1, and as we saw in each iteration the number of operations performed by the inner loop is k(n-i). Hence the total number of operations performed by this loop is

**k(n-0) + k(n-1) + k(n-2) + ... + k(n-(n-1)) = kn + k(n-1) + k(n-2) + ... + k(1) = k σ𝑛 𝑖 = kn(n+1)/2 = kn2/2 + kn/2. 𝑖=1**

Outside the outer loop a constant number k’ of additional operations are performed, so the total number of operations performed by the algorithm is **k’ + kn/2 + kn2/2**. the dominating term is **kn2/2**, so the time complexity of the algorithm is **O(n2).**

**Complexity:**

**1) O(1):**

O(1) represents an algorithm that takes the same amount of time to execute regardless of the number of inputs. So, 1 item takes 1 second, 10 items take 1 second, 100 items take 1 second and so on. Therefore performance is not affected by the size of the input data.

Time complexity of a function (or set of statements) is considered as O(1) if it doesn’t contain

loop, recursion and call to any other non-constant time function.

Set of non-recursive and non-loop statements. For example [swap() function](http://geeksquiz.com/c-program-swap-two-numbers/) has O(1) time complexity.  
 A loop or recursion that runs a constant number of times is also considered as O(1). For example the

following loop is O(1).

**for** (**int** i = 1; i <= c; i++)

{

// some O(1) expressions

}

**2) O(n):**

O(N) represents an algorithm where the size of the input data impacts the execution time. The performance of the algorithm is directly proportional to the number of inputs. So, 1 item takes 1 second, 10 items take 10 seconds, 100 items take 100 seconds and so on.

Time Complexity of a loop is considered as O(n) if the loop variables is incremented / decremented by

a constant amount. For example following functions have O(n) time complexity.

**for** (**int** i = 1; i <= n; i += c)

{

// some O(1) expressions

}

**OR**

**for** (**int** i = 1; i <= n; i -= c)

{

// some O(1) expressions

}

O(2N) represents an algorithm where execution time is doubled for each additional input. So 1 item takes 2 seconds, 2 items take 4 seconds, 3 items take 8 seconds and so on.

**3) O(n2)**:

 represents an algorithm that is directly proportional to the square of the sizes of the inputs and must performs N2calculations (by definition). Bubblesort is a good example of this algorithm.

Time complexity of nested loops is equal to the number of times the innermost statement is

executed. For example the following sample loops have **O(n2)** time complexity

**for** (**int** i = 1; i <=n; i += c)

{

**for** (**int** j = 1; j <=n; j += c)

{

// some O(1) expressions

}

}

**OR**

**for** (**int** i = n; i > 0; i -= c)

{

**for** (**int** j = i+1; j <=n; j += c)

{

// some O(1) expressions

}

}

For example [Selection sort](http://geeksquiz.com/selection-sort/) and [Insertion Sort](http://geeksquiz.com/insertion-sort/) have O(n2) time complexity.

**4) O(Logn)**:

O(log N) represents an algorithm where the number of computations grows linearly as input data grows exponentially. So 1 item takes 1 second, 10 items take 2 seconds, 100 items take 3 seconds and so on.

Time Complexity of a loop is considered as O(Logn) if the loop variables is divided /

multiplied by a constant amount.

**for** (**int** i = 1; i <=n; i \*= c)

{

// some O(1) expressions

}

**OR**

**for** (**int** i = n; i > 0; i /= c)

{

// some O(1) expressions

}

For example [Binary Search(refer iterative implementation)](http://geeksquiz.com/binary-search/) has O(Log n) time complexity. Let us see mathematically how it is O(Log n). The series that we get in first loop is 1, c, c2, c3, … ck. If we put k equals to Logcn, we get cLogcn which is n.

**5) O(LogLogn):**

(N2) represents an algorithm that will in increase in execution time proportionate to the number of the input times the logarithm of the number of the input. Mergesort and quicksort are good examples of this algorithm.

Time Complexity of a loop is considered as O(LogLogn) if the loop variables is reduced / increased exponentially by a constant amount.

// Here is C is constant greater than 1

**for** (**int** i = 2; i < n; i = pow(i, c))

{

// some O(1) expressions

}

**OR**

// Here is fun is sqrt or cube root or any other constant root

**for** (**int** i = n; i > 0; i = fun(i))

{

// some O(1) expressions

}

**O(1) time**   
1. Accessing Array Index (int a = ARR[5];)  
2. Inserting a node in Linked List  
3. Pushing and Poping on Stack  
4. Insertion and Removal from Queue  
5. Finding out the parent or left/right child of a node in a tree stored in Array  
6. Jumping to Next/Previous element in Doubly Linked List  
and you can find a million more such examples...

**O(n) time**  
1. Traversing an array  
2. Traversing a linked list  
3. Linear Search  
4. Deletion of a specific element in a Linked List (Not sorted)  
5. Comparing two strings  
6. Checking for Palindrome  
7. Counting/Bucket Sort  
and here too you can find a million more such examples....  
In a nutshell, all Brute Force Algorithms, or Noob ones which require linearity, are based on O(n) time complexity

**O(log n) time**  
1. Binary Search  
2. Finding largest/smallest number in a binary search tree  
3. Certain Divide and Conquer Algorithms based on Linear functionality  
4. Calculating Fibonacci Numbers - Best Method  
The basic premise here is NOT using the complete data, and reducing the problem size with every iteration

**O(nlogn) time**  
1. Merge Sort  
2. Heap Sort  
3. Quick Sort  
4. Certain Divide and Conquer Algorithms based on optimizing O(n^2) algorithms  
The factor of 'log n' is introduced by bringing into consideration Divide and Conquer. Some of these algorithms are the best optimized ones and used frequently.

**O(n^2) time**  
1. Bubble Sort  
2. Insertion Sort  
3. Selection Sort  
4. Traversing a simple 2D array

**Asymptotic Analysis:**

Following asymptotic notations are used to calculate the running time complexity of an algorithm.

* **O** − Big Oh
* **Ω** − Big omega
* **θ** − Big theta
* **o** − Little Oh
* **ω** − Little omega

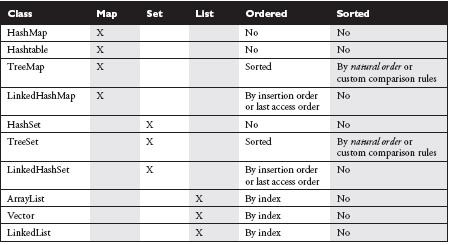
The asymptotic behavior of a function ***f(n)*** refers to the growth of ***f(n)*** as **n**gets large.

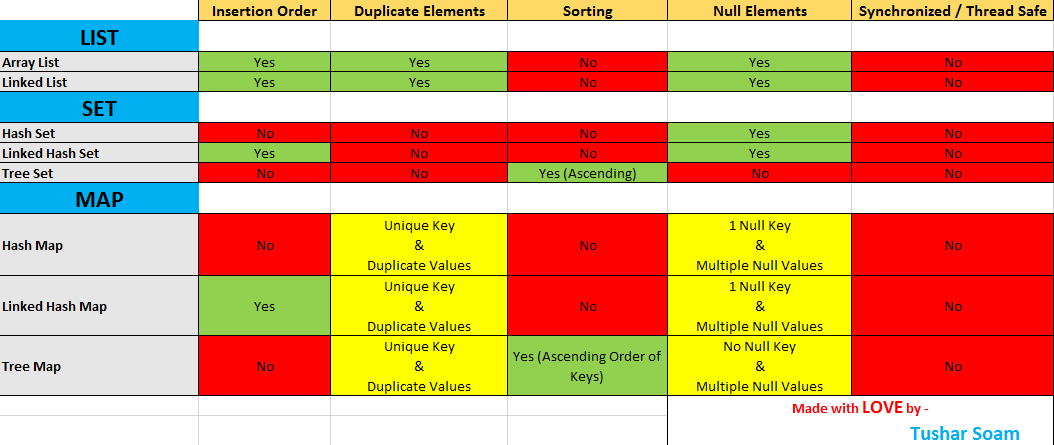
We typically ignore small values of **n**, since we are usually interested in estimating how slow the program will be on large inputs. A good rule of thumb is that the slower the asymptotic growth rate, the better the algorithm. Though it’s not always true.

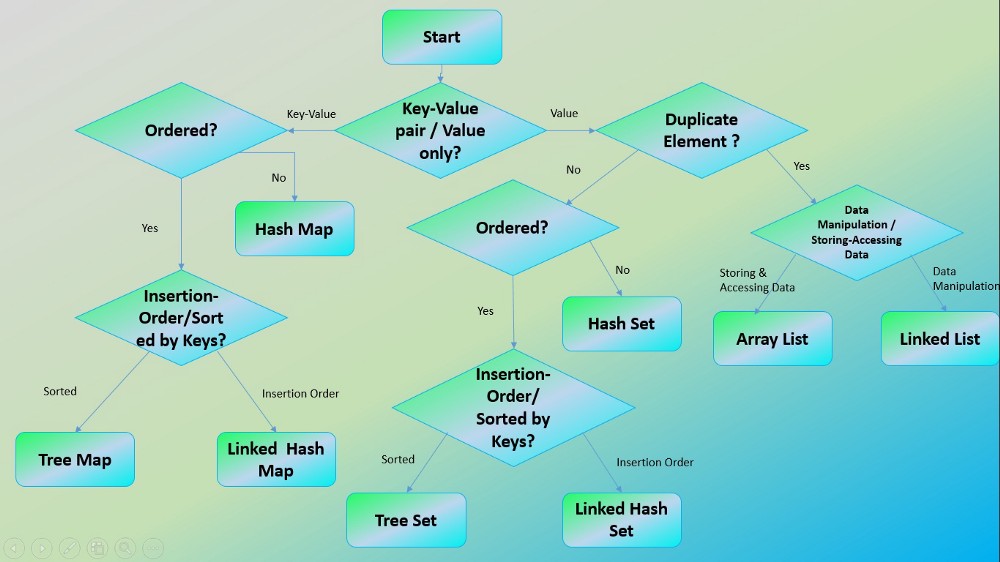
For example, a linear algorithm f(n)=d∗n+k is always asymptotically better than a quadratic one, f(n)=c.n2+q.

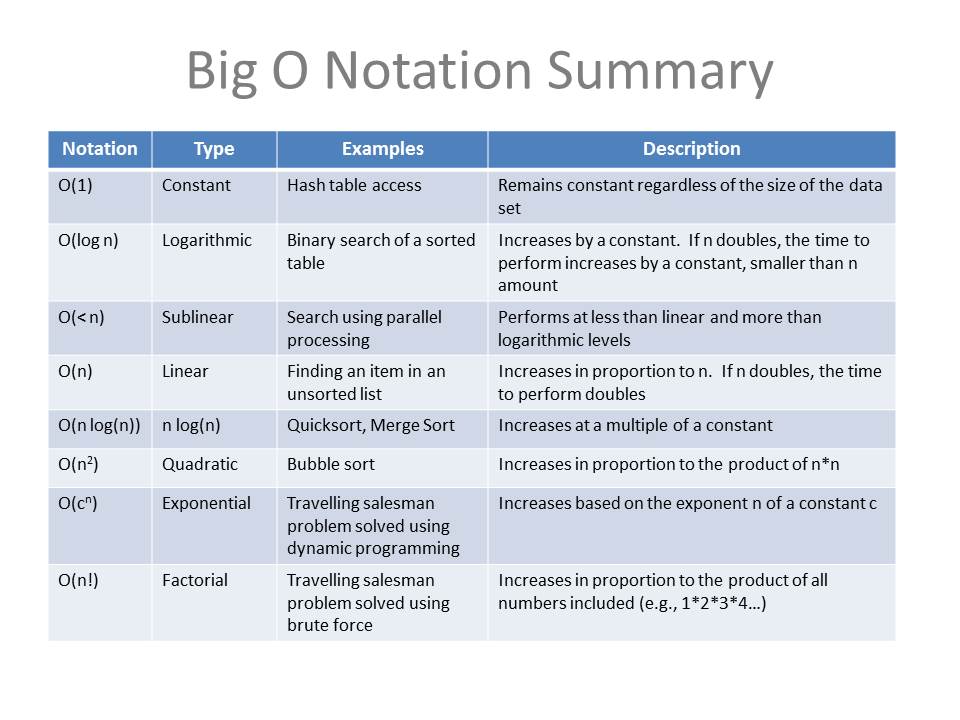
**Space efficiency and time efficiency:** They reach at two opposite ends. The more time efficiency you have, the less space efficiency you have and vice versa.

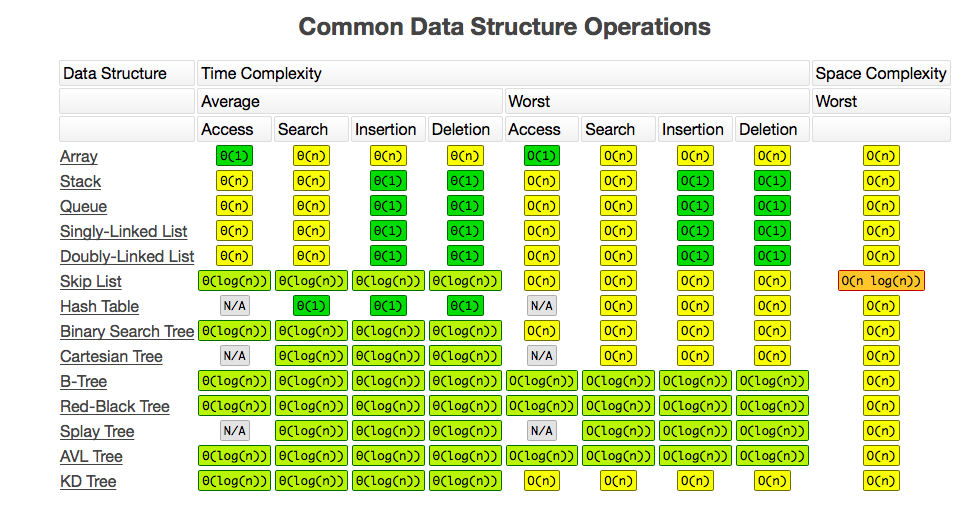
For example, Mergesort algorithm is exceedingly fast but requires a lot of space to do the operations. On the other side, Bubble Sort is exceedingly slow but requires the minimum space.











|  |
| --- |
|  |

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ **Searching Alogorithms** \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

<https://www.hackerearth.com/practice/algorithms/sorting/insertion-sort/visualize/>

<https://www.educative.io/collection/page/10370001/760001/1320001>

Searching Algorithms are designed to check for an element or retrieve an element from any data structure where it is stored. Based on the type of search operation, these algorithms are generally classified into two categories:

**Linear Search:**

1. **Sequential Search**: In this, the list or array is traversed sequentially and every element is checked. For example: [Linear Search](https://www.geeksforgeeks.org/linear-search/).



**The time complexity of above algorithm is O(n).**

Linear search is rarely used practically because other search algorithms such as the binary search algorithm and hash tables allow significantly faster searching comparison to Linear search.

|  |
| --- |
| **import** java.util.\*;  **import** java.util.\*;  **class** InterviewQuestion {  // Java code for linearly searching x in arr[]. If x  // is present then return its location, otherwise  **public** **static** **int** search(**int** arr[], **int** target) {  **for** (**int** i = 0; i < arr.length; i++) {  **if** (arr[i] == target) {  **return** i;  }  }  **return** -1;  }  **public** **static** **void** main(String args[]) {  **int** arr[] = { 2, 3, 4, 10, 40 };  **int** target = 10;  **int** result = *search*(arr, target);  **if** (result == -1)  System.***out***.println(" Elment is not present in Array ");  **else**  System.***out***.print("Element is present at index " + result);  }  } |

**Binary Search:**

1. **Interval Search**: These algorithms are specifically designed for searching in sorted data-structures. These type of searching algorithms are much more efficient than Linear Search as they repeatedly target the center of the search structure and divide the search space in half. For Example: [Binary Search](https://www.geeksforgeeks.org/binary-search/).



Time complexity of Binary search is **O(log n)**

The recurrence can be solved either using Recurrence Tree method or Master method.

|  |
| --- |
| **Recursive implementation of Binary Search:**  // Java implementation of recursive Binary Search  **class** InterviewQuestion{    // Returns index of x if it is present in arr[l..r], else return -1  **int** binarySearch(**int** arr[], **int** left, **int** right, **int** target)  {  **if** (right >= left) {  **int** mid = left + (right - left) / 2;    // If the element is present at the middle itself, return the target  **if** (arr[mid] == target)  **return** mid;    // If element is smaller than mid, then it can only be present in left subarray  **if** (arr[mid] > target)  **return** binarySearch(arr, left, mid - 1, target);    // Else the element can only be present in right subarray  **return** binarySearch(arr, mid + 1, right, target);  }    // We reach here when element is not present in array  **return** -1;  }    // Driver method to test above  **public** **static** **void** main(String args[])  {  //InterviewQuestion ob = new InterviewQuestion();  InterviewQuestion drive = **new** InterviewQuestion();    **int** arr[] = { 2, 3, 4, 10, 40 };  **int** n = arr.length;  **int** target = 10;  **int** result = drive.binarySearch(arr, 0, n - 1, target);  **if** (result == -1)  System.***out***.println("Element not present");  **else**  System.***out***.println("Element found at index " + result);  }  **Iterative implementation of Binary Search:**  **OR**  **int** binarySearch(**int** arr[], **int** x)  {  **int** l = 0, r = arr.length - 1;  **while** (l <= r) {  **int** m = l + (r - l) / 2;  // Check if x is present at mid  **if** (arr[m] == x)  **return** m;  // If x greater, ignore left half  **if** (arr[m] < x)  l = m + 1;  // If x is smaller, ignore right half  **else**  r = m - 1;  }  // if we reach here, then element was  // not present  **return** -1;  } |

**Jump Search:**

**jump search** (also known as the **block search**). Time complexity **O(√ n)**

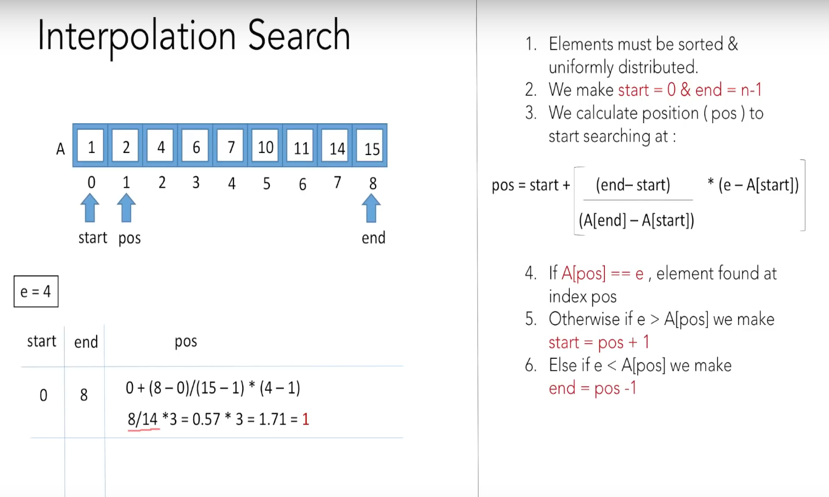
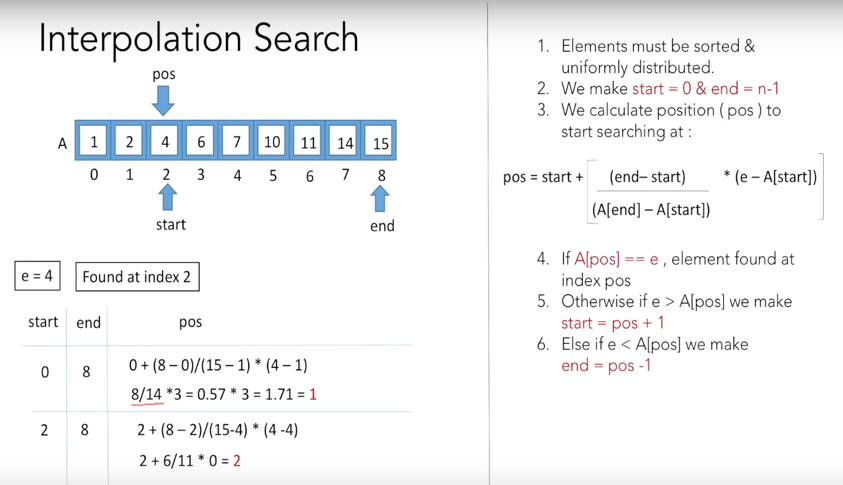
Like [Binary Search](http://geeksquiz.com/binary-search/), Jump Search is a searching algorithm for sorted arrays. The basic idea is to check fewer elements (than [linear search](https://www.geeksforgeeks.org/analysis-of-algorithms-set-2-asymptotic-analysis/)) by jumping ahead by fixed steps or skipping some elements in place of searching all elements.

|  |
| --- |
| // Java implementation of recursive Binary Search  **class** InterviewQuestion {  **public** **static** **int** jumpSearch(**int**[] arr, **int** x) {  **int** n = arr.length;  // Calculating the jump length over array elements  **int** step = (**int**) Math.*floor*(Math.*sqrt*(n));    // present (if it is present)  **int** prev = 0;    // Finding the block where element is (Attention! inside the array not the element)  **while** (arr[Math.*min*(step, n) - 1] < x) {  prev = step;  step += (**int**) Math.*floor*(Math.*sqrt*(n));  **if** (prev >= n)  **return** -1;  }  // Doing a linear search for x in block. Beginning with prev.  **while** (arr[prev] < x) {  prev++;  // If we reached next block or end of array, element is not present.  **if** (prev == Math.*min*(step, n))  **return** -1;  }  // If element is found  **if** (arr[prev] == x)  **return** prev;  **return** -1;  }  // Driver program to test function  **public** **static** **void** main(String[] args) {  **int** arr[] = { 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610 };  **int** x = 55;  // Find the index of 'x' using Jump Search  **int** index = *jumpSearch*(arr, x);  // Print the index where 'x' is located  System.***out***.println("\nNumber " + x + " is at index " + index);  }  } |

# **Interpolation Search:**

Time complexity: **O (log log n))**

The Interpolation Search is an improvement over [Binary Search](http://quiz.geeksforgeeks.org/binary-search/) for instances, where the values in a sorted array are uniformly distributed. Binary Search always goes to the middle element to check. On the other hand, interpolation search may go to different locations according to the value of the key being searched. For example, if the value of the key is closer to the last element, interpolation search is likely to start search toward the end side. To find the position to be searched, it uses following formula.

**Step1:** In a loop, calculate the value of “pos” using the probe position formula.  
**Step2:** If it is a match, return the index of the item, and exit.  
**Step3:** If the item is less than arr[pos], calculate the probe position of the left sub-array. Otherwise calculate the same in the right sub-array.  
**Step4:** Repeat until a match is found or the sub-array reduces to zero.

|  |
| --- |
| // Java implementation of recursive Binary Search  **class** InterviewQuestion {  // Array of items on which search will  // be conducted.  **static** **int** *arr*[] = **new** **int**[]{10, 12, 13, 16, 18, 19, 20, 21, 22, 23,  24, 33, 35, 42, 47};    // If x is present in arr[0..n-1], then returns index of it, else returns -1.  **static** **int** interpolationSearch(**int** x)  {  // Find indexes of two corners  **int** lo = 0, hi = (*arr*.length - 1);    // Since array is sorted, an element present in array must be in range defined by corner  **while** (lo <= hi && x >= *arr*[lo] && x <= *arr*[hi])  {  // Probing the position with keeping uniform distribution in mind.  **int** pos = lo + (((hi-lo) /  (*arr*[hi]-*arr*[lo]))\*(x - *arr*[lo]));    // Condition of target found  **if** (*arr*[pos] == x)  **return** pos;    // If x is larger, x is in upper part  **if** (*arr*[pos] < x)  lo = pos + 1;    // If x is smaller, x is in the lower part  **else**  hi = pos - 1;  }  **return** -1;  }    // Driver method  **public** **static** **void** main(String[] args)  {  **int** x = 18; // Element to be searched  **int** index = *interpolationSearch*(x);    // If element was found  **if** (index != -1)  System.***out***.println("Element found at index " + index);  **else**  System.***out***.println("Element not found.");  }  } |

# **Arrays.binarySearch() in Java:**

**Arrays.binarySearch()** is the simplest and most efficient method to find an element in a sorted array in Java

**Declaration:**

public static int binarySearch(data\_type arr, data\_type key )

public static int binarySearch(data\_type arr,range, range, data\_type key )

where **data\_type** can be any of the primitive data types: **byte**, **char**, **double**, **int**, **float**, **short**, **long**and **Object** as well.

**Description:**

# **Exponential Search:**

The name of this searching algorithm may be misleading as it works in O(Log n) time. The name comes from the way it searches an element.

**Time Complexity : O(Log n)**

1. Exponential Binary Search is particularly useful for unbounded searches, where size of array is infinite. Please refer [Unbounded Binary Search](https://www.geeksforgeeks.org/find-the-point-where-a-function-becomes-negative/) for an example.
2. It works better than Binary Search for bounded arrays, and also when the element to be searched is closer to the first element.

Exponential search involves two steps:

1. Find range where element is present
2. Do Binary Search in above found range.

**How to find the range where element may be present?**  
The idea is to start with subarray size 1, compare its last element with x, then try size 2, then 4 and so on until last element of a subarray is not greater.  
Once we find an index i (after repeated doubling of i), we know that the element must be present between i/2 and i (Why i/2? because we could not find a greater value in previous iteration)

Given below are the implementations of above steps.

|  |
| --- |
| // Java implementation of recursive Binary Search  **class** InterviewQuestion {  // Returns position of first occurrence of  // x in array  **static** **int** exponentialSearch(**int** arr[], **int** n, **int** x)  {  // If x is present at firt location itself  **if** (arr[0] == x)  **return** 0;    // Find range for binary search by repeated doubling  **int** i = 1;  **while** (i < n && arr[i] <= x)  i = i\*2;    // Call binary search for the found range.  **return** Arrays.*binarySearch*(arr, i/2, Math.*min*(i, n), x);  }    // Driver code  **public** **static** **void** main(String args[])  {  **int** arr[] = {2, 3, 4, 10, 40};  **int** x = 10;  **int** result = *exponentialSearch*(arr, arr.length, x);    System.***out***.println((result < 0) ?  "Element is not present in array" :  "Element is present at index " +  result);  }  } |

# **Sublist Search (Search a linked list in another list):**

Given two linked lists, the task is to check whether the first list is present in 2nd list or not.

Algorithm:  
1- Take first node of second list.  
2- Start matching the first list from this first node.  
3- If whole lists match return true.  
4- Else break and take first list to the first node again.  
5- And take second list to its second node.  
6- Repeat these steps until any of linked lists becomes empty.  
7- If first list becomes empty then list found else not.

Time Complexity : O(m\*n) where m is the number of nodes in second list and n in first.

|  |
| --- |
| **import** java.util.LinkedList;  **import** java.util.List;  // Java implementation of recursive Binary Search  **class** InterviewQuestion {  /\*  Given two linked list, you have to check whether the first one is present in2nd one or not?  Input : list1 = 10->20  list2 = 5->10->20  Output : LIST FOUND    Input : list1 = 1->2->3->4  list2 = 1->2->1->2->3->4  Output : LIST FOUND Input    : list1 = 1->2->3->4  list2 = 1->2->2->1->2->3  Output : LIST NOT FOUND    \*/  **private** **static** **int** interpolationSearch(List<Integer> list\_1, List<Integer> list\_2) {  **int** i = 0, j = i + 3, m = 0;  **while** (i < list\_1.size()) {  **if** (list\_1.get(i).equals(list\_2.get(m)) && list\_1.get(j).equals(list\_2.get(list\_2.size() - 1))) {  **while** (list\_1.get(i).equals(list\_2.get(m))) {  i++;  m++;  **if** (j + 1 < list\_1.size())  j++;  **if** (m == list\_2.size() - 1)  **return** 1;  }  **if** (!list\_1.get(i).equals(list\_2.get(m))) {  i++;  m = 0;  **if** (j + 1 < list\_1.size())  j++;  }  } **else** {  i++;  j++;  }  }  **return** -1;  }  // Driver method  **public** **static** **void** main(String[] args){  List<Integer> list\_1 = **new** LinkedList<Integer>();  List<Integer> list\_2 = **new** LinkedList<Integer>();  //put data into linked list as per your choice.  **int** index = *interpolationSearch*(list\_1, list\_2);  // If list\_2 is sublist of list\_1  **if** (index == -1)  System.***out***.println("LIST NOT FOUND");  **else**  System.***out***.println("LIST FOUND");  }  } |

**Fibonacci Search:** Complexity: **O(log(n))**

is a comparison-based technique that uses Fibonacci numbers to search an element in a sorted array.

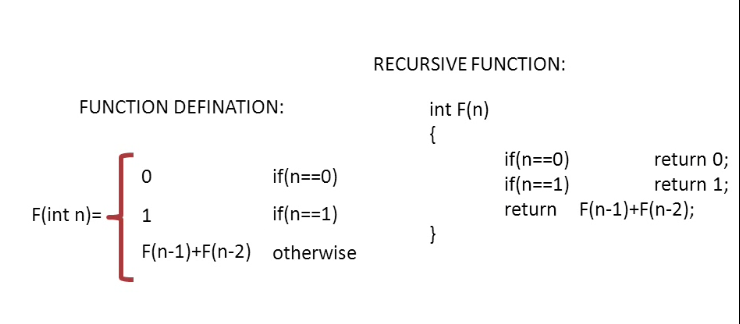
**Similarities with Binary Search:**

1. Works for sorted arrays
2. A Divide and Conquer Algorithm.
3. Has Log n time complexity.

**Differences with Binary Search**:

1. Fibonacci Search divides given array in unequal parts
2. Binary Search uses division operator to divide range. Fibonacci Search doesn’t use /, but uses + and -. The division operator may be costly on some CPUs.
3. Fibonacci Search examines relatively closer elements in subsequent steps. So when input array is big that cannot fit in CPU cache or even in RAM, Fibonacci Search can be useful.

Recursion Fibonacci. This recursive method rpeated calculation, so avoid using this recursive method.



|  |
| --- |
| **// Fibonacci number**  **class** InterviewQuestion {    // Fibonacci using recursion. Some steps repeats  **public** **static** **int** fibonacciRecursive(**int** n) {  **if** ( n == 1)  **return** 1;  **if** (n == 2)  **return** 1;  **else**    **return** *fibonacciRecursive*(n-1) + *fibonacciRecursive*(n-2);  }    // Fibonacci using iteration. Benefit, has less space complexity.  **public** **static** **int** fibonacciIteration(**int** n){  **int** x = 0, y = 1, z = x +y;  **for** (**int** i= 1; i < n; i++) {  x = y;  y = z;  z = x + y;  }  **return** y;  }    // Driver  **public** **static** **void** main(String arg[]) {    **int** n = 5;  System.***out***.println("Iteration");  **for** (**int** i = 1; i <= n; i++) {    System.***out***.print(*fibonacciIteration*(i) + " >> ");  }  System.***out***.println( " \nRecursive ");  **for** (**int** i = 1; i <= n; i++) {    System.***out***.print(*fibonacciRecursive*(i) + " >> ");  }    }  Output: Iteration  1 >> 1 >> 2 >> 3 >> 5 >>  Recursive  1 >> 1 >> 2 >> 3 >> 5 >> |

|  |
| --- |
| **// Java Fibonacci Search**  **class** InterviewQuestion {    // Utility function to find minimum of two elements  //public static int min(int x, int y)  //{ return (x <= y)? x : y; }  // Returns index of x if present, else returns -1  **public** **static** **int** fibMonaccianSearch(**int** arr[], **int** x, **int** n)  {  // Initialize fibonacci numbers  **int** fibn1 = 0; // (m-2)'th Fibonacci No.  **int** fibn2 = 1; // (m-1)'th Fibonacci No  **int** TotalF = fibn1 + fibn2; // m'th Fibonacci    // TotalF is going to store the smallest Fibonacci Number greater than or equal to n  **while** (TotalF < n)  {  fibn1 = fibn2;  fibn2 = TotalF;  TotalF = fibn1 + fibn2;  }    // Marks the eliminated range from front  **int** offset = -1;    // while there are elements to be inspected.Note that we compare arr[fibn1] with x.  // When TotalF becomes 1, fibn1 becomes 0  **while** (TotalF > 1)  {  // Check if fibn1 is a valid location  **int** i = Math.*min*(offset+fibn1, n-1);    // If x is greater than the value at index fibn1, cut the subarray array from offset to i  **if** (arr[i] < x)  {  TotalF = fibn2;  fibn2 = fibn1;  fibn1 = TotalF - fibn2;  offset = i;  }    // If x is greater than the value at index fibn1, cut the subarray after i+1  **else** **if** (arr[i] > x)  {  TotalF = fibn1;  fibn2 = fibn2 - fibn1;  fibn1 = TotalF - fibn2;  }    // element found return index  **else** **return** i;  }    // comparing the last element with x  **if**(fibn2 == 1 && arr[offset+1] == x)  **return** offset+1;    // element not found. return -1  **return** -1;  }    // driver code  **public** **static** **void** main(String[] args)  {  **int** arr[] = {10, 22, 35, 40, 45, 50,  80, 82, 85, 90, 100};  **int** n = arr.length;  **int** x = 85;  **int** index = *fibMonaccianSearch*(arr, x, n);  **if** ( index == -1)  System.***out***.println("Error Value of X not Found!");  **else**  System.***out***.print ("Found at index: "+ *fibMonaccianSearch*(arr, x, n));  }  } |

**Ternary Search: Time Complexity O(log3N**

Like linear search and binary search, ternary search is a searching technique that is used to determine the position of a specific value in an array. In binary search, the sorted array is divided into two parts while in ternary search, it is divided into 3 parts and then you determine in which part the element exists.

Ternary search, like binary search, is a divide-and-conquer algorithm. It is mandatory for the array (in which you will search for an element) to be sorted before you begin the search. In this search, after each iteration it neglects ⅓⅓ part of the array and repeats the same operations on the remaining ⅔⅔.

|  |
| --- |
| **int** ternary\_search(**int** l,**int** r, **int** x)  {  **if**(r>=l)  {  **int** mid1 = l + (r-l)/3;  **int** mid2 = r - (r-l)/3;  **if**(ar[mid1] == x)  **return** mid1;  **if**(ar[mid2] == x)  **return** mid2;  **if**(x<ar[mid1])  **return** ternary\_search(l,mid1-1,x);  **else** **if**(x>ar[mid2])  **return** ternary\_search(mid2+1,r,x);  **else**  **return** ternary\_search(mid1+1,mid2-1,x);  }  **return** -1;  } |

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ **Sorting Alogorithms** \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Selection Sort: Tme Complexity: О(n2)** as there are two nested loops.

The selection sort algorithm sorts an array by repeatedly finding the minimum element (considering ascending order) from unsorted part and putting it at the beginning. The algorithm maintains two subarrays in a given array.

1) The subarray which is already sorted.  
2) Remaining subarray which is unsorted.

In every iteration of selection sort, the minimum element (considering ascending order) from the unsorted subarray is picked and moved to the sorted subarray.

Following example explains the above steps:

|  |
| --- |
| **import** java.util.Arrays;  **public** **class** InterviewQuestion {  **public** **static** **void** SelectionSort(**int** arr[]) {  **int** n = arr.length;  // One by one move boundary of unsorted subarray  **for** (**int** i = 0; i < n - 1; i++) {  // Find the minimum element in unsorted array  **for** (**int** j = i + 1; j < n; j++)  **if** (arr[j] < arr[i])  {    // Swap the found minimum element with the first  // element  **int** temp = arr[i];  // I always save the minimum value  arr[i] = arr[j];  arr[j] = temp;  }  }  System.***out***.println("Sorted Array");  System.***out***.println(Arrays.*toString*(arr));  }  // Driver code to test above  **public** **static** **void** main(String args[]) {  **int** arr[] = { 64, 25, 12, 22, 11 };  System.***out***.println("Un Sorted Array");  System.***out***.println(Arrays.*toString*(arr));  System.***out***.println( " ");  *SelectionSort*(arr);  }  } |

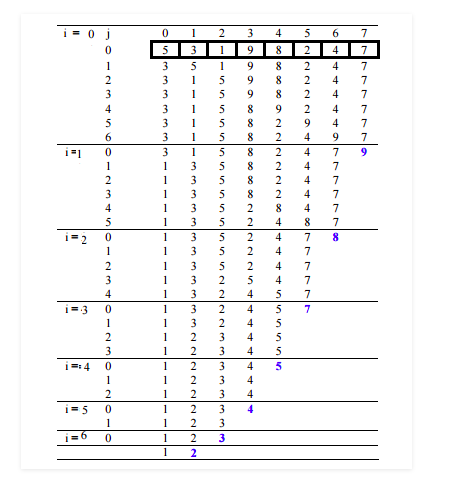
**Bubble Sort: Tme Complexity: О(n2)**

Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in wrong order.

**Example:**  
**First Pass:**  
( **5** **1** 4 2 8 ) –> ( **1** **5** 4 2 8 ), Here, algorithm compares the first two elements, and swaps since 5 > 1.  
( 1 **5** **4** 2 8 ) –>  ( 1 **4** **5** 2 8 ), Swap since 5 > 4  
( 1 4 **5** **2** 8 ) –>  ( 1 4 **2** **5** 8 ), Swap since 5 > 2  
( 1 4 2 **5** **8** ) –> ( 1 4 2 **5** **8** ), Now, since these elements are already in order (8 > 5), algorithm does not swap them.

**Second Pass:**  
( **1** **4** 2 5 8 ) –> ( **1** **4** 2 5 8 )  
( 1 **4** **2** 5 8 ) –> ( 1 **2** **4** 5 8 ), Swap since 4 > 2  
( 1 2 **4** **5** 8 ) –> ( 1 2 **4** **5** 8 )  
( 1 2 4 **5** **8** ) –>  ( 1 2 4 **5** **8** )  
Now, the array is already sorted, but our algorithm does not know if it is completed. The algorithm needs one **whole** pass without **any** swap to know it is sorted.

**Third Pass:**  
( **1** **2** 4 5 8 ) –> ( **1** **2** 4 5 8 )  
( 1 **2** **4** 5 8 ) –> ( 1 **2** **4** 5 8 )  
( 1 2 **4** **5** 8 ) –> ( 1 2 **4** **5** 8 )  
( 1 2 4 **5** **8** ) –> ( 1 2 4 **5** **8** )



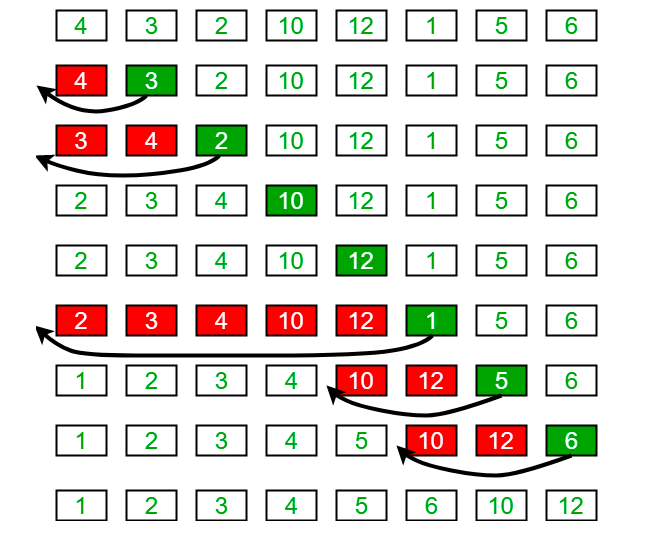
|  |
| --- |
| **import** java.util.Arrays;  **public** **class** InterviewQuestion {  **public** **static** **void** bubbleSort(**int** arr[]) {  **int** n = arr.length;    **for** (**int** i = 0; i < n - 1; i++) {  **for** (**int** j = 0; j < n - i - 1; j++) {  **if** (arr[j] > arr[j + 1]) {  // swap arr[j+1] and arr[i] int temp = arr[j];  **int** temp = arr[j];  arr[j] = arr[j + 1];  arr[j + 1] = temp;  }  }  }  }  /\* Prints the array \*/  **void** printArray(**int** arr[]) {  **int** n = arr.length;  **for** (**int** i = 0; i < n; ++i)  System.***out***.print(arr[i] + " ");  System.***out***.println();  }  // Driver method to test above  **public** **static** **void** main(String args[]) {  **int** arr[] = { 5, 3, 1, 9, 8, 2, 4, 7 };  System.***out***.println(Arrays.*toString*(arr));  System.***out***.println(" ");    *bubbleSort*(arr);  System.***out***.println("Sorted array");  System.***out***.println(Arrays.*toString*(arr));  }  } |

**Insertion Sort: Tme Complexity: О(n2)**

Insertion sort is a simple sorting algorithm that works the way we sort playing cards in our hands.

**Algorithm**  
// Sort an arr[] of size n  
insertionSort(arr, n)  
Loop from i = 1 to n-1.  
……a) Pick element arr[i] and insert it into sorted sequence arr[0…i-1]

**Example:**



|  |
| --- |
| // Java program for implementation of Insertion Sort  **class** InterviewQuestion {  /\* Function to sort array using insertion sort \*/  **public** **static** **void** sort(**int** arr[]) {  **int** n = arr.length;  **for** (**int** i = 1; i < n; ++i) {  **int** key = arr[i];  **int** j = i - 1;  /\*  \* Move elements of arr[0..i-1], that are greater than key, to one position  \* ahead of their current position  \*/  **while** (j >= 0 && arr[j] > key) {  arr[j + 1] = arr[j];  j = j - 1;  }  arr[j + 1] = key;  }  }  /\* A utility function to print array of size n \*/  **static** **void** printArray(**int** arr[]) {  **int** n = arr.length;  **for** (**int** i = 0; i < n; ++i)  System.***out***.print(arr[i] + " ");  System.***out***.println();  }  // Driver method  **public** **static** **void** main(String args[]) {  **int** arr[] = { 12, 11, 13, 5, 6 };  *sort*(arr);  *printArray*(arr);  }  } |

# **Merge Sort: Time complexity: O(nlogn)**

**Note: whenever you call recursively inside the brace, anything in the brace execute the recursive call times.**

**Void Sort(x, y)** {

If (x>y)

{

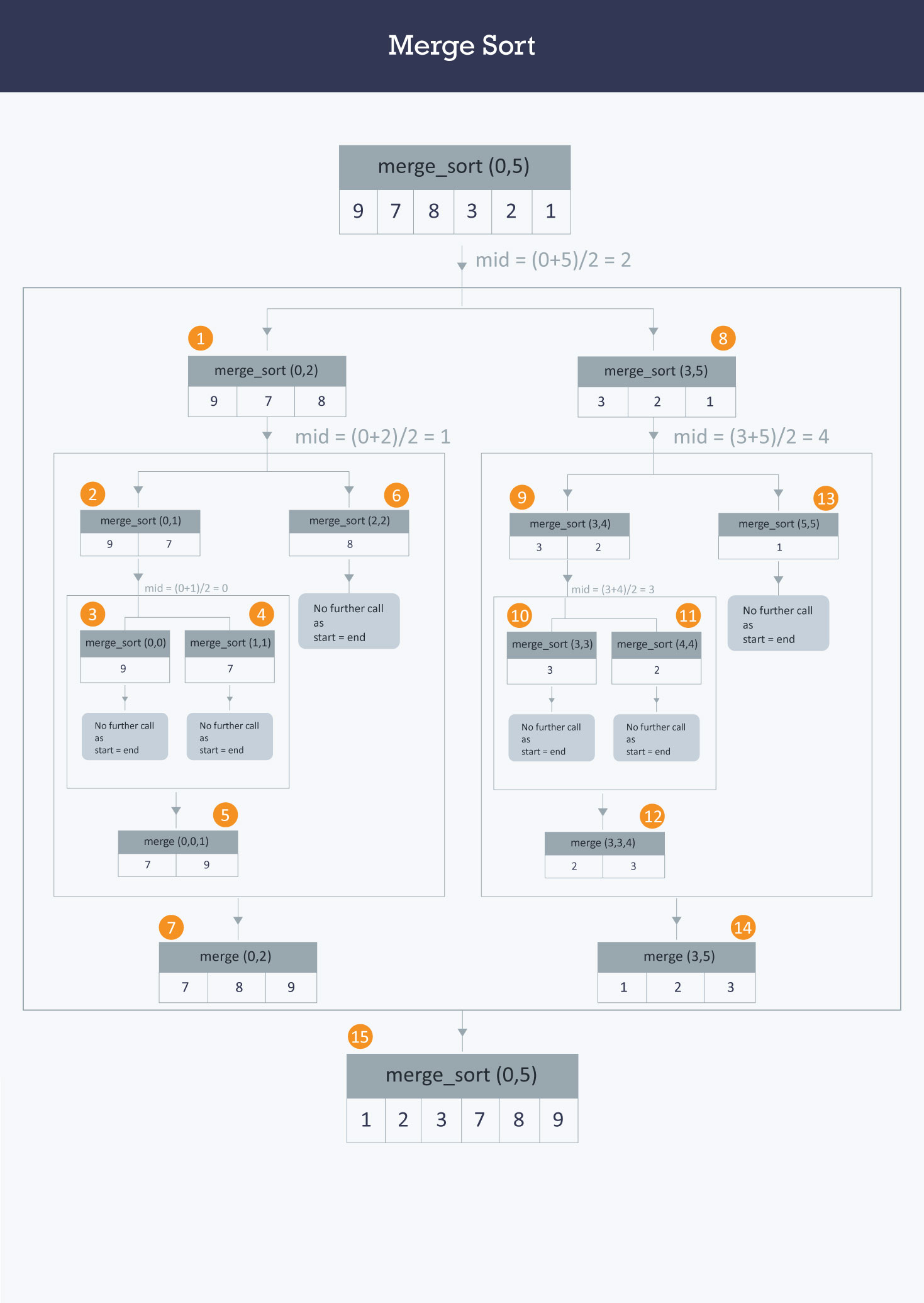
Sort(x, y);

}

Sysout(“x”); // execute recursive times

}

Like [QuickSort](https://www.geeksforgeeks.org/quick-sort/), Merge Sort is a [Divide and Conquer](https://www.geeksforgeeks.org/divide-and-conquer-introduction/) algorithm. It divides input array in two halves, calls itself for the two halves and then merges the two sorted halves. **The merge() function** is used for merging two halves. The merge(arr, l, m, r) is key process that assumes that arr[l..m] and arr[m+1..r] are sorted and merges the two sorted sub-arrays into one.



|  |
| --- |
| **package** Test;  **import** java.util.Arrays;  **public** **class** mergeSorting{  **public** **static** **void** mergeSort(**int** arr[]) {  **if** (arr.length > 1) {  **int** m = arr.length / 2;  **int**[] L = **new** **int**[m];  **int**[] R = **new** **int**[arr.length - m];  // Coping value to Right array  **for** (**int** i = 0; i < R.length; i++) {  R[i] = arr[m + i];  }    System.***out***.println("The value of Righ Array" + Arrays.*toString*(R));  // Coping value to Left array  **for** (**int** j = 0; j < m; j++) {  L[j] = arr[j];  }  *mergeSort*(L);  *mergeSort*(R);  *merge*(arr, L, R);  }  }  **public** **static** **void** merge(**int**[] arr, **int**[] L, **int**[] R) {  **int** i = 0, j = 0, k = 0;  **while** (i < L.length && j < R.length) {  **if** (L[i] < R[j]) {  arr[k] = L[i];  i++;  } **else** {  arr[k] = R[j];  j++;  }  k++;  }  **while** (i < L.length) {  arr[k] = L[i];  i++;  k++;  }  **while** (j < R.length) {  arr[k] = R[j];  j++;  k++;  }  }  // Driver method to test above  **public** **static** **void** main(String args[]) {  **int** arr[] = { 5, 10, 6, 7, 11, 13, 1, 4, 2 };  *mergeSort*(arr);  **for** (**int** i = 0; i < arr.length; i++) {  System.***out***.println("Sorted array " + arr[i]);  }  }  } |

OR

|  |
| --- |
| mport java.lang.reflect.Array;  import java.util.Arrays;  // Java program for implementation of Insertion Sort  class InterviewQuestion {  public static int[] ArrayDivider(int[] arry) {  if (arry.length <= 1) {  return arry;  }  // Declear new left and right array  int[] leftArray;  int[] rightArray;  // Divide the Array into equal part & save it to left  int mid = arry.length / 2;  leftArray = new int[mid];  rightArray = new int[arry.length - mid];  // Copy the Array to the left & right side of new Array  for (int i = 0; i < mid; i++) {  leftArray[i] = arry[i];  }  for (int j = 0; j < rightArray.length; j++) {  rightArray[j] = arry[mid + j];  }  // Create new Array to hold sorted right and left Array  int[] result = new int[arry.length];  // Call & save left & right array  leftArray = ArrayDivider(leftArray);  rightArray = ArrayDivider(rightArray);  // Call to sort & save the array  result = meregeSort(leftArray, rightArray);  return result;  }  // To merege and sort the array  public static int[] meregeSort(int[] leftArray, int[] rightArray) {  int[] result = new int[leftArray.length + rightArray.length];  int leftPointer, rightPointer, resultPointer;  leftPointer = rightPointer = resultPointer = 0;  // To check if there is element in one side of Array  while (leftPointer < leftArray.length || rightPointer < rightArray.length) {  // To check if there is elements in right and left Arrays  if (leftPointer < leftArray.length && rightPointer < rightArray.length) {  // if left array < right array, copy left array to result array or copy right  // array  if (leftArray[leftPointer] < rightArray[rightPointer]) {  result[resultPointer++] = leftArray[leftPointer++];  } else {  result[resultPointer++] = rightArray[rightPointer++];  }  }  //  else if (leftPointer < leftArray.length) {  result[resultPointer++] = leftArray[leftPointer++];  } else if (rightPointer < rightArray.length) {  result[resultPointer++] = rightArray[rightPointer++];  }  }  return result;  }  // Driver method  public static void main(String args[]) {  int[] arry = { 3, 8, 10, 6, 15, 2, 9 };  System.out.println("Unsorted Array");  System.out.println(Arrays.toString(arry));  arry = ArrayDivider(arry);  System.out.println("Sorted Array: ");  System.out.println(Arrays.toString(arry));  }  } |

|  |
| --- |
| **//Recursion Merge Sorting**    **class** InterviewQuestion  {  // Merges two subarrays of arr[].  // First subarray is arr[l..m]  // Second subarray is arr[m+1..r]  **void** merge(**int** arr[], **int** l, **int** m, **int** r)  {  // Find sizes of two subarrays to be merged  **int** n1 = m - l + 1;  **int** n2 = r - m;    /\* Create temp arrays \*/  **int** L[] = **new** **int** [n1];  **int** R[] = **new** **int** [n2];    /\*Copy data to temp arrays\*/  **for** (**int** i=0; i<n1; ++i)  L[i] = arr[l + i];  **for** (**int** j=0; j<n2; ++j)  R[j] = arr[m + 1+ j];      /\* Merge the temp arrays \*/    // Initial indexes of first and second subarrays  **int** i = 0, j = 0;    // Initial index of merged subarry array  **int** k = l;  **while** (i < n1 && j < n2)  {  **if** (L[i] <= R[j])  {  arr[k] = L[i];  i++;  }  **else**  {  arr[k] = R[j];  j++;  }  k++;  }    /\* Copy remaining elements of L[] if any \*/  **while** (i < n1)  {  arr[k] = L[i];  i++;  k++;  }    /\* Copy remaining elements of R[] if any \*/  **while** (j < n2)  {  arr[k] = R[j];  j++;  k++;  }  }    // Main function that sorts arr[l..r] using  // merge()  **void** sort(**int** arr[], **int** l, **int** r)  {  System.***out***.println("Test Print l: " + l);  **if** (l < r)  {  // Find the middle point  **int** m = (l+r)/2;    System.***out***.println("Test Print l after division : " + l);  // Sort first and second halves  sort(arr, l, m);  sort(arr , m+1, r);    // Merge the sorted halves  merge(arr, l, m, r);  }  }    /\* A utility function to print array of size n \*/  **static** **void** printArray(**int** arr[])  {  **int** n = arr.length;  **for** (**int** i=0; i<n; ++i)  System.***out***.print(arr[i] + " ");  System.***out***.println();  }    // Driver method  **public** **static** **void** main(String args[])  {  **int** arr[] = {12, 11, 13, 5, 6, 7};    System.***out***.println("Given Array");  *printArray*(arr);    InterviewQuestion ob = **new** InterviewQuestion();  ob.sort(arr, 0, arr.length-1);    System.***out***.println("\nSorted array");  *printArray*(arr);  }  } |

|  |
| --- |
| **// Iteration Merge Soriting**  **import** java.util.Arrays;  **public** **class** InterviewQuestion {  **public** **static** **void** mergeSort(**int**[] array) {  **if** (array == **null**) {  **return**;  }  **if** (array.length > 1) {  **int** mid = array.length / 2;  // Split left part  **int**[] left = **new** **int**[mid];  **for** (**int** i = 0; i < mid; i++) {  left[i] = array[i];  }  // Split right part  **int**[] right = **new** **int**[array.length - mid];  **for** (**int** i = mid; i < array.length; i++) {  right[i - mid] = array[i];  }  *mergeSort*(left);  *mergeSort*(right);  *sorting*(array, left, right);  }  }  **public** **static** **void** sorting(**int**[] array, **int**[] left, **int**[] right) {  **int** i = 0;  **int** j = 0;  **int** k = 0;  // Merge left and right arrays  **while** (i < left.length && j < right.length) {  **if** (left[i] < right[j]) {  array[k] = left[i];  i++;  } **else** {  array[k] = right[j];  j++;  }  k++;  }  // Collect remaining elements  **while** (i < left.length) {  array[k] = left[i];  i++;  k++;  }  **while** (j < right.length) {  array[k] = right[j];  j++;  k++;  }  }  // Driver program to test above functions.  **public** **static** **void** main(String[] args) {  **int** arr[] = { 12, 11, 13, 5, 6, 7 };  **int** i = 0;  System.***out***.println("Given array is");  **for** (i = 0; i < arr.length; i++)  System.***out***.print(arr[i] + " ");  *mergeSort*(arr);  System.***out***.println("\n");  System.***out***.println("Sorted array is");  **for** (i = 0; i < arr.length; i++)  System.***out***.print(arr[i] + " ");  }  } |

# QuickSort:

Like [Merge Sort](http://quiz.geeksforgeeks.org/merge-sort/), QuickSort is a Divide and Conquer algorithm. It picks an element as pivot and partitions the given array around the picked pivot.  There are many different versions of quickSort that pick pivot in different ways.

1. Always pick first element as pivot.
2. Always pick last element as pivot (implemented below)
3. Pick a random element as pivot.
4. Pick median as pivot.

The key process in quickSort is partition(). Target of partitions is, given an array and an element x of array as pivot, put x at its correct position in sorted array and put all smaller elements (smaller than x) before x, and put all greater elements (greater than x) after x. All this should be done in linear time.

|  |
| --- |
| **// Quick Sort**  **import** java.util.Arrays;  **public** **class** InterviewQuestion {  /\*  \* This function takes last element as pivot, places the pivot element at its  \* correct position in sorted array, and places all smaller (smaller than pivot)  \* to left of pivot and all greater elements to right of pivot  \*/  **public** **static** **int** partition(**int** arr[], **int** low, **int** high) {  **int** pivot = arr[high];    **int** i = (low - 1); // index of smaller element  **for** (**int** j = low; j < high; j++) {  // If current element is smaller than or  // equal to pivot  System.***out***.println("Test :" + Arrays.*toString*(arr));    **if** (arr[j] <= pivot) {    i++;  // swap arr[i] and arr[j]  **int** temp = arr[i];  arr[i] = arr[j];  arr[j] = temp;    }  }  // swap arr[i+1] and arr[high] (or pivot)  **int** temp = arr[i + 1];  arr[i + 1] = arr[high];  arr[high] = temp;  **return** i + 1;  }  /\*  \* The main function that implements QuickSort() arr[] --> Array to be sorted,  \* low --> Starting index, high --> Ending index  \*/  **public** **static** **void** sort(**int** arr[], **int** low, **int** high) {  **if** (low < high) {  /\*  \* pi is partitioning index, arr[pi] is now at right place  \*/  **int** pi = *partition*(arr, low, high);  // Recursively sort elements before  // partition and after partition  *sort*(arr, low, pi - 1);  *sort*(arr, pi + 1, high);  }  }  // Driver program  **public** **static** **void** main(String args[]) {  **int** arr[] = { 10, 7, 8, 9, 1, 5 };  System.***out***.println(Arrays.*toString*(arr));  **int** n = arr.length;  *sort*(arr, 0, n - 1);  System.***out***.print("sorted array: ");  System.***out***.println(Arrays.*toString*(arr));  }  } |

OR

|  |
| --- |
| **package** Test;  **import** java.util.Arrays;  **public** **class** quickSort {  // Parition the array  **public** **static** **void** partition(**int**[] arr, **int** low, **int** high) {  **if** (low >= high) {  **return**;  }  // Save the last index form the sorted array in to index  **int** index = *sort*(arr, low, high);  // Call recursivelly to divide the array  *partition*(arr, low, index - 1);  *partition*(arr, index, high);  }  **public** **static** **int** sort(**int**[] arr, **int** low, **int** high) {  // The middle value of array  **int** pivot = (low + high) / 2;  // Loop until left is equal to right array  **while** (low <= high) {  // Compare left array with pivot and increment low  **while** (arr[low] < arr[pivot]) {  low++;  }  // Compare right array with pivot and increment high  **while** (arr[high] > arr[pivot]) {  high--;  }  // Swap left value with right value if left greater than right  **if** (low <= high) {  **int** temp = arr[low];  arr[low] = arr[high];  arr[high] = temp;  low++;  high--;  }  }  // return  **return** low;  }  // Driver program  **public** **static** **void** main(String args[]) {  **int** arr[] = { 5, 7, 11, 9, 20, 15 };  **int** n = arr.length - 1;  *partition*(arr, 0, n);  **for** (**int** i = 0; i < arr.length; i++) {  System.***out***.print(arr[i] + " ");  }  }  } |

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ **Data structure Solving Problems** \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ **Finding Duplicate numbers** \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

|  |
| --- |
| **Finding Repeating number:**  **import** java.util.Arrays;  **class** test {  /// Function to print duplicates  **void** printRepeating(**int** arr[], **int** size)  {  **int** i;  System.***out***.println("The repeating elements are : ");    **for** (i = 0; i < size; i++)  {  **if** (arr[Math.*abs*(arr[i])] >= 0)  arr[Math.*abs*(arr[i])] = -arr[Math.*abs*(arr[i])];  **else**  System.***out***.print(Math.*abs*(arr[i]) + " ");  }  }    // Driver program  **public** **static** **void** main(String[] args)  {  test duplicate = **new** test();  **int** arr[] = {1, 2, 3, 1, 3, 6, 6};  **int** arr\_size = arr.length;    duplicate.printRepeating(arr, arr\_size);  }  } |

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ **Finding Duplicate characters** \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

|  |
| --- |
| **Method 1:**  // Author Jeremiah Haile  // Solved used model-based technique.  // Minimum value is known, So we can use model-based technique  **public** **class** StringSortingLexologicaly {  **public** **static** **void** findDuplicateChar(String ss) {  **boolean**[] bl = **new** **boolean**[26];  **for** (**int** i = 0; i < ss.length(); i++) {  String co = ss.toLowerCase();  **int** result = co.charAt(i) - 'a';  // To ignore space  **if** (result >= 0) {  // To handle just alphabet  **if** (bl[result]) {  // result is integer so we cast into char to get character  **char** ch = (**char**) (result + 'a');  System.***out***.println("Duplicated Char is: " + ch);  } **else** {  bl[result] = **true**;  }  }  }  }  **public** **static** **void** main(String args[]) {  String s = "Programming is great!";  System.***out***.println();  *findDuplicateChar*(s);  }  }  Output:  Duplicated Char is: r  Duplicated Char is: m  Duplicated Char is: g  Duplicated Char is: i  Duplicated Char is: g  Duplicated Char is: r  Duplicated Char is: a |

|  |
| --- |
| **Method 2:**  **package** Test;  **import** java.util.Arrays;  **class** test {  **public** **static** **void** main(String[] args) {  String str = "abcdabghplhhnfl".toLowerCase();  // create a integer array for 26 alphabets.  // where index 0,1,2.. will be the container for frequency of a,b,c...  Integer[] ar = **new** Integer[26];  // fill the integer array with character frequency.  **for** (**int** i = 0; i < str.length(); i++) {  **int** j = str.charAt(i) - 'a';  **if** (ar[j] == **null**) {  ar[j] = 1;  } **else** {  ar[j] += 1;  }  }  // print only those alphabets having frequency greater then 1.  **for** (**int** i = 0; i < ar.length; i++) {  **if** (ar[i] != **null** && ar[i] > 1) {  **char** c = (**char**) (97 + i);  System.***out***.println("'" + c + "' comes " + ar[i] + " times.");  }  }  }  }  Output:  'a' comes 2 times.  'b' comes 2 times.  'h' comes 3 times.  'l' comes 2 times. |

|  |
| --- |
| **Method 3:**  // Author Jeremiah Haile  // Solved using HashMap.  **import** java.util.\*;  **class** DuplicaCharacterSearch {  **public** **static** **void** duplicate(String s) {  Map<Character, Integer> map = **new** HashMap<Character, Integer>();  **int** i = 0;  **while** (i < s.length()) {  **char** ch = s.charAt(i);  // To skip white space  **if** (ch > 32) {  **if** (map.containsKey(ch)) {  map.put(ch, map.get(ch) + 1);  } **else** {  map.put(ch, 1);  }  }  i++;  }  // Loop using (Object name : map.name()) method over the key and value  **for** (Object keyName : map.keySet()) {  **if** (map.get(keyName) > 1) {  System.***out***.print(keyName);  System.***out***.println(" duplicated " + map.get(keyName) + " times");  }  }  }  **public** **static** **void** main(String args[]) {  String s = "Programming is great!";  System.***out***.println();  *duplicate*(s);  }  }  Output:  a duplicated 2 times  r duplicated 3 times  g duplicated 3 times  i duplicated 2 times  m duplicated 2 times |

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ **Sorting String Alphabetically** \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

|  |
| --- |
| **package** Test;  **import** java.util.Arrays;  **import** java.util.Scanner;  // Author Jeremiah Haile  // Solved used brute force technique.  **public** **class** test {  **public** **static** String stringSort(String ss) {  String[] s = ss.split(" ");  // System.out.println(Arrays.toString(s));  **for** (**int** j = 0; j < s.length; j++) {  **for** (**int** i = j + 1; i < s.length; i++) {  // String s = "string" s.charAt(); works fine here. However, it won't work if we  // have String array  // the same way instead we use specific string index -> String[] s = {  // "aff","ass", "ass"} -> s[i].charAt(0);  // It compare character by character and give the difference in number  **if** (s[j].compareTo(s[i]) > 0) {  *swap*(i, j, s);  }  }  }  **return** Arrays.*toString*(s);  // System.out.println(Arrays.toString(s));  }  // To swap  **public** **static** **void** swap(**int** i, **int** j, String[] s) {  String temp = s[j];  s[j] = s[i];  s[i] = temp;  }  **public** **static** **void** main(String args[]) {  String s = "Apple Banana Orange Bluebery Pich";  System.***out***.println();  String g = *stringSort*(s);  System.***out***.println(g);  // System.out.println(result);  }  } |

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ **Finding Missing numbers** \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

|  |
| --- |
| **Method 1:**  **This program work for Multiple missing number**  **Time complexity O(n) and memory complexity O(n)**  // Author Jeremiah Haile  // Solved using model-based technique.  // Minimum value is known, So we can use model-based  **public** **class** test {  **public** **static** **void** main(String[] args) {  **int** arr[] = { 20, 6, 10, 5, 7 };  **int** difference = 0;  **int** result = 0;  Arrays.*sort*(arr);  System.***out***.println(Arrays.*toString*(arr));  // Specify the size of temp array.  **int** max = (arr[arr.length - 1] - arr[0]);  **int**[] temp = **new** **int**[max + 1];  // Copy value of arr to corresponding temp array  **for** (**int** i = 0; i < arr.length - 1; i++) {    // To save the first arr value  **if** (i == 0) {  temp[0] = arr[0];  }  difference = arr[i + 1] - arr[i];  result = result + difference;  temp[result] = arr[i + 1];  }  **for** (**int** j = 0; j < temp.length - 1; j++) {  **if** (temp[j] == 0) {  **int** tem = temp[j - 1] + 1;  temp[j] = tem;  System.***out***.println(" Missing Number: " + tem);  }  }  }  } |

|  |
| --- |
| **Method 2:**  **This Algorithm only work for one missing number**  **Time complexity O(n)**  Algorithm: this only work for one missing number. It may has overflow issue  1. Get the sum of numbers  total = n\*(n+1)/2  2 Subtract all the numbers from sum and  you will get the missing number. |

|  |
| --- |
| **Method 3:**  **This Algorithm only work for one missing number**  **Time complexity O(n) and memory complexity O(1)**  **Using Xor**  This approach is based on the XOR operation, remember:  A B | A XOR B 0 0 | 0 0 1 | 1 1 0 | 1 1 1 | 0  so: A ^ 0 = A A ^ A = 0 A ^ B = C C ^ A = B  **import** java.util.Arrays;  **class** InterviewQuestion {  // Function to find missing number  **static** **int** getMissingNo(**int** a[], **int** n) {  **int** XOR = 0;  **int** XOS = 0;  **for** (**int** i = 0; i <= a.length - 1; i++) {  XOS ^= a[i];  }  **for** (**int** i = 1; i <= n; i++) {  XOS ^= i;  }  **return** (XOR ^ XOS);  }  /\* program to test above function \*/  **public** **static** **void** main(String args[]) {  **int** a[] = { 2, 3, 5, 4, 6 };  System.***out***.println(Arrays.*toString*(a));  **int** j = 6;  **int** miss = *getMissingNo*(a, j);  System.***out***.println(miss);  }  } |

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ **Finding the sum of two numbers** \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

|  |
| --- |
| **Finding the sum of two integer number in array: This can be done using Brut Furce techniq**  **But that has O(n^2) efficiency.**  **Method 1:**  **Time complexity O(n)**  **import** java.util.Arrays;  **import** java.util.HashSet;  **class** InterviewQuestion {    **static** **void** printpairs(**int** arr[],**int** sum)  {  HashSet<Integer> s = **new** HashSet<Integer>();  **for** (**int** i=0; i<arr.length; ++i)  {  **int** temp = sum-arr[i];    // checking for condition  **if** (temp>=0 && s.contains(temp))  {  System.***out***.println("Pair with given sum " + sum + " is (" + arr[i] + ", "+temp+")");  }  s.add(arr[i]);  }  }    // Main to test the above function  **public** **static** **void** main (String[] args)  {  **int** A[] = {1, 4, 45, 6, 10, 8};  **int** n = 16;  *printpairs*(A, n);  }  } |

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ **Check if a String contains numbers** \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

|  |
| --- |
| **Method 1:**  // Author Jeremiah Haile  **import** java.util.regex.Pattern;  **public** **class** test {  **public** **static** **void** containsNumber(String str) {  **for** (**int** i = 0; i < str.length(); i++) {  **char** x = str.charAt(i);  **int** y = (**int**) x;  **if** (y < 97) {  System.***out***.println("True, it contains number at index " + i);  **break**;  } **else** {  System.***out***.println("Not ture, it doesn'tcontain number!");  **break**;  }  }  }  // Driver code  **public** **static** **void** main(String[] args) {  String str = "77hhhhhg777";  *containsNumber*(str);  }  } |

|  |
| --- |
| **Method 2:**  **public** **boolean** containsOnlyDigits(String str) {  **if** (str.isEmpty())  **return** **false**;  **for** (**int** i = 0; i < str.length(); i++) {  // (int 49 - 57) represent char 0 - 9  **char** temp = (**char**) (str.charAt(i) - '0');  **if** (temp > 0 && temp < 9)  **return** **true**;  }  **return** **false**;  } |

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ **Reverse a String** \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

|  |
| --- |
| **Method 1:**    **static** **void** reverse()  {    String input = "Geeks For Geeks";    // convert String to character array  **char**[] try1 = input.toCharArray();    **for** (**int** i = try1.length-1; i>=0; i--)  System.***out***.print(try1[i]);  } |

|  |
| --- |
| **Method 2:**  **static** **public** **void** reverse()  {  String input = "Geeks For Geeks";    StringBuilder input1 = **new** StringBuilder();    // append a string into StringBuilder input1  input1.append(input);    // reverse StringBuilder input1  input1 = input1.reverse();    // print reversed String  System.***out***.println(input1);  } |

|  |
| --- |
| **Method 2:**  **package** Test;  // Author Jeremiah Haile  // Solved using recursive call technique.  **public** **class** test {  **public** **static** **void** reverseString(String str) {  **if** (str.length() <= 0) {  **return**;  }    **char** re = str.charAt(str.length() - 1);    System.***out***.print(re);    // Recursive call using substring method  *reverseString*(str.substring(0, (str.length() - 1)));  } |

\_\_\_\_\_\_\_\_\_ **Swap two variables without using temp variables** \_\_\_\_\_\_\_\_\_

**E.g x = 10, y = 5, to x = 5, y = 10:**

|  |
| --- |
| **class** Geeks {  **public** **static** **void** main(String a[])      {  **int** x = 10;  **int** y = 5;          x = x + y;          y = x - y;          x = x - y;          System.out.println("After swaping:"  + " x = " + x + ", y = " + y);      }  } |

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_­­­­\_\_\_\_\_\_\_\_\_ **Reverse a number** \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**E.g n = 123 to n = 321**

|  |
| --- |
| **class** GFG  {      Iterative function to reverse digits of num  **static** **int** reversDigits(**int** num)      {  **int** rev\_num = 0;  **while**(num > 0)          {              rev\_num = rev\_num \* 10 + num % 10;              num = num / 10;          }  **return** rev\_num;      }        // Driver code  **public** **static** **void** main (String[] args)      {  **int** num = 4562;          System.out.println("Reverse of no. is "  + reversDigits(num));      }  } |

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_­­­­\_\_\_\_\_\_\_\_\_\_\_ **Count vawels** \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

|  |
| --- |
| **package** Test;  **import** java.util.\*;  **import** java.util.regex.Pattern;  **public** **class** test {  Map<Character, Integer> map = **new** LinkedHashMap<>();  **public** **void** containVowels(String str) {  **for** (**int** i = 0; i < str.length(); i++) {  str.toLowerCase();  **char** x = str.charAt(i);  **if** (x == 'a' || x == 'i' || x == 'o' || x == 'u' || x == 'e') {  **if** (map.containsKey(x)) {  map.put(x, map.get(x) + 1);  } **else** {  // String s = Character.toString(x);  map.put(x, 1);  }  }  }  **for** (Object key : map.keySet()) {  // System.out.println("The value of vowel " + );  **if** ((**char**) key == 'a' || (**char**) key == 'i' || (**char**) key == 'o' || (**char**) key ==  'u' || (**char**) key == 'e') {  System.***out***.println("The vowel " + key + " comes " + map.get(key) + " times!");  }  }  }  // Driver code  **public** **static** **void** main(String[] args) {  String str = "Jeremiah";  test t = **new** test();  t.containVowels(str);  }  } |

Iterating using Map.Entry<>

|  |
| --- |
| // Iterating using Map.Entry<Objext, Object>  **for** (Map.Entry<Character, Integer> key : map.entrySet()) {  // System.out.println("The value of vowel " + );  **if** (key.getKey() == 'a' || key.getKey() == 'i' || key.getKey() == 'o' || key.getKey() == 'u'  || key.getKey() == 'e') {  System.***out***.println("The vowel " + key.getKey() + " comes " + key.getValue() + " times!");  }  }  } |

|  |
| --- |
| **Method 2:**  **package** Test;  // Author Jeremiah Haile  // Solved using switch statement.  **import** java.util.\*;  **public** **class** test {  **int** a = 0, i = 0, o = 0, u = 0, e = 0;  **public** **void** countingVowels(String str) {  **for** (**int** j = 0; j < str.length(); j++) {  str.toLowerCase();  **char** x = (**char**) str.charAt(j);  **switch** (x) {  **case** 'a':  a += 1;  *printv*('a', a);  **break**;  **case** 'i':  i += 1;  test.*printv*('i', i);  **break**;  **case** 'o':  o += 1;  test.*printv*('o', o);  **break**;  **case** 'u':  u += 1;  test.*printv*('e', u);  **break**;  **case** 'e':  e += 1;  test.*printv*('e', e);  **break**;  **default**:  **break**;  }  }  // System.out.println("a comes: " + a + " i comes: " + i + " o comes: " + o + "  // e comes: " + e + " u comes: " + u);  }  **public** **static** **void** printv(**char** v, **int** n) {  System.***out***.println(" Vowel " + v + " comes " + n + " times!");  }  // Driver code  **public** **static** **void** main(String[] args) {  String str = "Jaremiah";  test t = **new** test();  t.countingVowels(str);  }  } |

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_­­­­\_\_\_ **Find Uppercase Characters** \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

|  |
| --- |
| // Author Jeremiah Haile  // Solved using linear search and isUpperCase() method.  **public** **void** countingVowels(String str) {  **for** (**int** i = 0; i < str.length(); i++) {  **if** (Character.*isUpperCase*(str.charAt(i))) {  System.***out***.print(str.charAt(i) + ",");  }  }  } |

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_­­­­\_\_\_\_\_\_ **Find Largest numbers** \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

|  |
| --- |
| // Author Jeremiah Haile  // Solved using linear search and Integer.***MIN\_VALUE***.  **int** large = Integer.***MIN\_VALUE***;  **public** **int** largestValue(**int**[] arr) {    **for** (**int** i = 0; i < arr.length; i++) {  **if** (arr[i] > large) {  large = arr[i];  }  }  **return** large;  } |

\_\_\_\_\_\_\_ **Check if a String is a valid shuffle of two String?** \_\_\_\_\_\_\_

|  |
| --- |
| **package** Test;  // Three declared variables used in while loop to compare. Then we compare the value of those  // variable according to their incrementation.  **public** **class** test {  // Returns true if C is an interleaving of A and B, otherwise returns false  **static** **boolean** isInterleaved(String A, String B, String C) {  **int** i = 0, j = 0, k = 0;  **while** (k != C.length() - 1) {  **if** (A.charAt(i) == C.charAt(k))  i++;  **else** **if** (B.charAt(j) == C.charAt(k))  j++;  // If doesn't match with either A or B, then return false  **else**  **return** **false**;  k++;  }  // If there is element in either A or B, return false  **if** (i < A.length() || j < B.length())  **return** **false**;  **return** **true**;  }  **public** **static** **void** main(String[] args) {  String A = "AB";  String B = "CD";  String C = "ACBG";  **if** (*isInterleaved*(A, B, C) == **true**)  System.***out***.printf("%s is interleaved of %s and %s", C, A, B);  **else**  System.***out***.printf("%s is not interleaved of %s and %s", C, A, B);  }  } |

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ **Remove a given character from String?** \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

|  |
| --- |
| **public** **class** test {  **public** **static** **void** remove(String st, **char** x) {  **for** (**int** i = 0; i < st.length(); i++) {  **char** s = st.charAt(i);  **if** (s == x) {  String ss = (st.substring(0, i) + st.substring(i + 1));  System.***out***.println("The string after char moved is " + ss);  }  }  }  **public** **static** **void** main(String[] args) {  **char** x = 'D';  String st = "ABGCD";  *remove*(st, x);  }  } |

\_\_\_\_\_\_\_\_\_\_\_\_\_ **Finding the longest palindromic substring** \_\_\_\_\_\_\_\_\_\_\_\_\_

|  |
| --- |
| **package** Test;  **public** **class** test {  **public** String longestPalindrome(String s) {  **if** (s.isEmpty()) {  **return** **null**;  }  **if** (s.length() == 1) {  **return** s;  }  String longest = s.substring(0, 1);  **for** (**int** i = 0; i < s.length(); i++) {  // get longest palindrome with center of i  String tmp = helper(s, i, i);  **if** (tmp.length() > longest.length()) {  longest = tmp;  }  // get longest palindrome with center of i, i+1  tmp = helper(s, i, i + 1);  **if** (tmp.length() > longest.length()) {  longest = tmp;  }  }  **return** longest;  }  |A|B|C|B|A|C|  |begin --> <-- end|  <--- (-) (+) --->  // Given a center, either one letter or two letter find longest palindrome    **public** String helper(String s, **int** begin, **int** end) {  **while** (begin >= 0 && end <= s.length() - 1 && s.charAt(begin) == s.charAt(end)) {  begin--;  end++;  }  **return** s.substring(begin + 1, end);  }  **public** **static** **void** main(String[] args) {  String s = "ABCBAC";  // System.out.println(s.substring(1, (s.length() - 1)));  test t = **new** test();  System.***out***.println(t.longestPalindrome(s));  }  } |

\_\_\_\_\_\_\_\_\_ **Sorting String in their length( Ascending form)** \_\_\_\_\_\_\_\_\_\_

|  |
| --- |
| **import** java.util.Arrays;  **import** java.util.Collections;  **import** java.util.Comparator;  **public** **class** test {  **public** **static** **void** main(String[] args) {  String S = "disentangle one could No correctly";  String W[] = S.split(" ");  Arrays.*sort*(W, **new** Comparator<String>() {  @Override  **public** **int** compare(String s1, String s2) {  // **TODO**: Argument validation (nullity, length)  **return** s1.length() - s2.length();// comparision  }  });  System.***out***.println(" " + Arrays.*toString*(W));  // return s1.length() - s2.length();// comparision  }  } |

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ **Find permutation of String** \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

|  |
| --- |
| **package** Test;  **public** **class** test {  **static** **void** printPermutn(String str, String ans) {  // If string is empty  **if** (str.length() == 0) {  System.***out***.print(ans + " ");  **return**;  }  **for** (**int** i = 0; i < str.length(); i++) {  **char** ch = str.charAt(i);  // Rest of the string after excluding the ith character  String ros = str.substring(0, i) + str.substring(i + 1);  // Recurvise call "abc"  // printPermutn(bc, a)  // printPermutn(c, ab)  // printPermutn( , abc)    // printPermutn(b, ac)  // printPermutn(, acb)  // printPermutn(ac, b)    // printPermutn(c, ba)  // printPermutn(, bac)  // printPermutn(a, bc)    // printPermutn(, bca)  // printPermutn(ab, c)  // printPermutn(b, ca)    // printPermutn(, cab)  // printPermutn(a, cb)  // printPermutn(, cba)  *printPermutn*(ros, ans + ch);  }  }  // Driver code  **public** **static** **void** main(String[] args) {  String s = "abb";  System.***out***.println("Permutation of " + s + " is: ");  *printPermutn*(s, "");  }  }  Output:  abc acb bac bca cab cba |

|  |
| --- |
| **Method 2** |

|  |
| --- |
| **Method 2**  **Package** Test;  //Java program to print all permutations of the given string  **import** java.util.\*;  **import** java.lang.\*;  **import** java.io.\*;  **public** **class** test {  **private** **char**[] arrA;  **public** **void** permutation(**char**[] arrA, **int** left, **int** size) {  **int** x;  **if** (left == size) {  **for** (**int** i = 0; i < arrA.length; i++) {  System.***out***.print(arrA[i]);  }  System.***out***.print(" ");  } **else** {  **for** (x = left; x < size; x++) {  swap(arrA, left, x);  permutation(arrA, left + 1, size);  swap(arrA, left, x);  }  }  }  **public** **void** swap(**char**[] arrA, **int** i, **int** j) {  **char** temp = arrA[i];  arrA[i] = arrA[j];  arrA[j] = temp;  }  **public** **static** **void** main(String[] args) **throws** java.lang.Exception {  String s = "abc";  **char**[] arrCh = s.toCharArray();  test i = **new** test();  i.permutation(arrCh, 0, arrCh.length);  }  } |

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ **BFS and DFS Search** \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

|  |
| --- |
| **package** Test;  **public** **class** Node {  String data;  Node left;  Node right;    **public** Node(String data) {  **this**.data = data;  }  **public** Node(String data, Node left, Node right) {  **this**.data = data;  **this**.left = left;  **this**.right = right;  }  } |

|  |
| --- |
| **package** Test;  **import** java.util.LinkedList;  **import** java.util.Queue;  **public** **class** test {  // BFS Traversal  **public** **static** **void** BFS\_traversal(Node node) {  Queue<Node> q = **new** LinkedList<Node>();  // Add trees first  q.add(node);  // Loop in to tree  **while** (!q.isEmpty()) {  node = q.remove();  System.***out***.print(node.data + " ");  **if** (node.left != **null**) {  q.add(node.left);  }  **if** (node.right != **null**) {  q.add(node.right);  }  }  }  // DFS Traversal  **public** **static** **void** DFS\_traversal(Node node) {  **if** (node == **null**) {  **return**;  }  System.***out***.print(node.data + " ");  *DFS\_traversal*(node.left);  *DFS\_traversal*(node.right);  }  **public** **static** **void** main(String[] args) **throws** java.lang.Exception {  // Populating the tree  Node tree = **new** Node("A", **new** Node("B", **new** Node("C"), **new** Node("D")),  **new** Node("E", **new** Node("F"), **new** Node("G", **new** Node("H"), **null**)));  System.***out***.println("BFS Output: ");  *BFS\_traversal*(tree);  System.***out***.println(" ");  System.***out***.println("\nDFS Output: ");  *DFS\_traversal*(tree);  }  } |

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ **Counting the Island Problem** \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

|  |
| --- |
| **package** Test;  **public** **class** Counting\_Iland {  **public** **int** numIslands(**char**[][] grid) {  **int** H = grid.length;  **int** W = grid[0].length;  **int** count = 0;  **for** (**int** i = 0; i < H; i++) {  **for** (**int** j = 0; j < W; j++) {  **if** (grid[i][j] == '1') {    // Always check to the top and left  **if** ((i == 0 || grid[i - 1][j] == '0') && (j == 0 || grid[i][j - 1] == '0')) {  count++;  }  }  }  }  **return** count;  }  **public** **static** **void** main(String args[]) {  **char**[][] grid = { { '0', '1', '1', '1', '0' },  { '0', '0', '1', '0', '0' },  { '0', '0', '0', '0', '0' },  { '1', '0', '1', '0', '0' },  { '1', '0', '1', '1', '1' } };  Counting\_Iland c = **new** Counting\_Iland();  System.***out***.println("There are " + c.numIslands(grid) + " Islands!");  }  } |

\_\_\_\_\_\_\_\_\_\_\_ **Find all palindrome permutations of a string** \_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ **Java Script** \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

|  |
| --- |
| <https://www.youtube.com/watch?v=hdI2bqOjy3c>  // ATTENTION: THIS IS CODE FROM THE YOUTUBE CRASH COURSE. IT IS NOT MEANT TO RUN, IT IS JUST FOR LEARNING PURPOSES //  // LOGGING OUTPUT  alert('Hello World'); // Do not use for debugging. Stops script and only strings  console.log('Hello World');  console.error('This is an error');  console.warn('This is a warning');  // VARIABLES - var, let, const  let age = 30;  // let can be re-assigned, const can not  age = 31;  // DATA TYPES - String, Number, Boolean, null, undefined  const name = 'Brad';  const age = 37;  const rating = 3.5;  const isCool = true;  const x = null;  const y = undefined;  let z; // undefined  // Check type  console.log(typeof z);  // STRINGS  // Concatenation  console.log('My name is ' + name + ' and I am ' + age);  // Template literal (better)  console.log(`My name is ${name} and I am ${age}`);  // String methods & properties  const s = 'Hello World';  let val;  // Get length  val = s.length;  // Change case  val = s.toUpperCase();  val = s.toLowerCase();  // Get sub string  val = s.substring(0, 5);  // Split into array  val = s.split('');    // ARRAYS - Store multiple values in a variable  const numbers = [1,2,3,4,5];  const fruits = ['apples', 'oranges', 'pears', 'grapes'];  console.log(numbers, fruit);  // Get one value - Arrays start at 0  console.log(fruits[1]);  // Add value  fruits[4] = 'blueberries';  // Add value using push()  fruits.push('strawberries');  // Add to beginning  fruits.unshift('mangos');  // Remove last value  fruits.pop();  // // Check if array  console.log(Array.isArray(fruits));  // // Get index  console.log(fruits.indexOf('oranges'));    // OBJECT LITERALS  const person = {  firstName: 'John',  age: 30,  hobbies: ['music', 'movies', 'sports'],  address: {  street: '50 Main st',  city: 'Boston',  state: 'MA'  }  }  // Get single value  console.log(person.name)  // Get array value  console.log(person.hobbies[1]);  // Get embedded object  console.log(person.address.city);  // Add property  person.email = 'jdoe@gmail.com';  // Array of objects  const todos = [  {  id: 1,  text: 'Take out trash',  isComplete: false  },  {  id: 2,  text: 'Dinner with wife',  isComplete: false  },  {  id: 3,  text: 'Meeting with boss',  isComplete: true  }  ];  // Get specific object value  console.log(todos[1].text);  // Format as JSON  console.log(JSON.stringify(todos));  // LOOPS  // For  for(let i = 0; i <= 10; i++){  console.log(`For Loop Number: ${i}`);  }  // While  let i = 0  while(i <= 10) {  console.log(`While Loop Number: ${i}`);  i++;  }  // Loop Through Arrays  // For Loop  for(let i = 0; i < todos.length; i++){  console.log(` Todo ${i + 1}: ${todos[i].text}`);  }  // For...of Loop  for(let todo of todos) {  console.log(todo.text);  }  // HIGH ORDER ARRAY METHODS (show prototype)  // forEach() - Loops through array  todos.forEach(function(todo, i, myTodos) {  console.log(`${i + 1}: ${todo.text}`);  console.log(myTodos);  });  // map() - Loop through and create new array  const todoTextArray = todos.map(function(todo) {  return todo.text;  });  console.log(todoTextArray);  // filter() - Returns array based on condition  const todo1 = todos.filter(function(todo) {  // Return only todos where id is 1  return todo.id === 1;  });  // CONDITIONALS  // Simple If/Else Statement  const x = 30;  if(x === 10) {  console.log('x is 10');  } else if(x > 10) {  console.log('x is greater than 10');  } else {  console.log('x is less than 10')  }  // Switch  color = 'blue';  switch(color) {  case 'red':  console.log('color is red');  case 'blue':  console.log('color is blue');  default:  console.log('color is not red or blue')  }  // Ternary operator / Shorthand if  const z = color === 'red' ? 10 : 20;    // FUNCTIONS  function greet(greeting = 'Hello', name) {  if(!name) {  // console.log(greeting);  return greeting;  } else {  // console.log(`${greeting} ${name}`);  return `${greeting} ${name}`;  }  }  // ARROW FUNCTIONS  const greet = (greeting = 'Hello', name = 'There') => `${greeting} ${name}`;  console.log(greet('Hi'));  // OOP  // Constructor Function  function Person(firstName, lastName, dob) {  // Set object properties  this.firstName = firstName;  this.lastName = lastName;  this.dob = new Date(dob); // Set to actual date object using Date constructor  // this.getBirthYear = function(){  // return this.dob.getFullYear();  // }  // this.getFullName = function() {  // return `${this.firstName} ${this.lastName}`  // }  }  // Get Birth Year  Person.prototype.getBirthYear = function () {  return this.dob.getFullYear();  }  // Get Full Name  Person.prototype.getFullName = function() {  return `${this.firstName} ${this.lastName}`  }  // Instantiate an object from the class  const person1 = new Person('John', 'Doe', '7-8-80');  const person2 = new Person('Steve', 'Smith', '8-2-90');  console.log(person2);  // console.log(person1.getBirthYear());  // console.log(person1.getFullName());    // Built in constructors  const name = new String('Kevin');  console.log(typeof name); // Shows 'Object'  const num = new Number(5);  console.log(typeof num); // Shows 'Object'  // ES6 CLASSES  class Person {  constructor(firstName, lastName, dob) {  this.firstName = firstName;  this.lastName = lastName;  this.dob = new Date(dob);  }  // Get Birth Year  getBirthYear() {  return this.dob.getFullYear();  }  // Get Full Name  getFullName() {  return `${this.firstName} ${this.lastName}`  }  }  const person1 = new Person('John', 'Doe', '7-8-80');  console.log(person1.getBirthYear());  // ELEMENT SELECTORS  // Single Element Selectors  console.log(document.getElementById('my-form'));  console.log(document.querySelector('.container'));  // Multiple Element Selectors  console.log(document.querySelectorAll('.item'));  console.log(document.getElementsByTagName('li'));  console.log(document.getElementsByClassName('item'));  const items = document.querySelectorAll('.item');  items.forEach((item) => console.log(item));  // MANIPULATING THE DOM  const ul = document.querySelector('.items');  // ul.remove();  // ul.lastElementChild.remove();  ul.firstElementChild.textContent = 'Hello';  ul.children[1].innerText = 'Brad';  ul.lastElementChild.innerHTML = '<h1>Hello</h1>';  const btn = document.querySelector('.btn');  // btn.style.background = 'red';  // EVENTS  // Mouse Event  btn.addEventListener('click', e => {  e.preventDefault();  console.log(e.target.className);  document.getElementById('my-form').style.background = '#ccc';  document.querySelector('body').classList.add('bg-dark');  ul.lastElementChild.innerHTML = '<h1>Changed</h1>';  });  // Keyboard Event  const nameInput = document.querySelector('#name');  nameInput.addEventListener('input', e => {  document.querySelector('.container').append(nameInput.value);  });  // USER FORM SCRIPT  // Put DOM elements into variables  const myForm = document.querySelector('#my-form');  const nameInput = document.querySelector('#name');  const emailInput = document.querySelector('#email');  const msg = document.querySelector('.msg');  const userList = document.querySelector('#users');  // Listen for form submit  myForm.addEventListener('submit', onSubmit);  function onSubmit(e) {  e.preventDefault();    if(nameInput.value === '' || emailInput.value === '') {  // alert('Please enter all fields');  msg.classList.add('error');  msg.innerHTML = 'Please enter all fields';  // Remove error after 3 seconds  setTimeout(() => msg.remove(), 3000);  } else {  // Create new list item with user  const li = document.createElement('li');  // Add text node with input values  li.appendChild(document.createTextNode(`${nameInput.value}: ${emailInput.value}`));  // Add HTML  // li.innerHTML = `<strong>${nameInput.value}</strong>e: ${emailInput.value}`;  // Append to ul  userList.appendChild(li);  // Clear fields  nameInput.value = '';  emailInput.value = '';  }  } |