Core Java – Interview Questions

OOPs, Collection, Multithreading

Version: 1.00

Revision History

Date	*Version*	*Description*	*Author*
06/02/2017	01	Release 1.0	Sunil Soni

Table of Contents

1.String vs StringBuffer vs StringBuilder

1.1. String in Java

- String class represents character strings, we can instantiate String by two ways.
  String str = "abc"; or String str = new String ("abc");
- String is immutable in java, so its easy to share it across different threads or functions.
- When we create a String using double quotes, it first looks for the String with same value in the JVM string pool, if found it returns the reference else it creates the String object and then place it in the String pool. This way JVM saves a lot of space by using same String in different threads. But if new operator is used, it explicitly creates a new String in the heap memory.
- + operator is overloaded for String and used to concatenate two Strings. Although internally it uses StringBuffer to perform this action.
- String overrides equals() and hashCode() methods, two Strings are equal only if they have same characters in same order. Note that equals() method is case sensitive, so if you are not looking for case sensitive checks, you should use equalsIgnoreCase() method.
- A String represents a string in the UTF-16 format
- String is a final class with all the fields as final except “private int hash”. This field contains the hashCode() function value and created only when hashCode() method is called and then cached in this field. Furthermore, hash is generated using final fields of String class with some calculations, so every time hashCode() method is called, it will result in same output. For caller, its like calculations are happening every time but internally it’s cached in hash field.

1.2. Difference between String and StringBuffer

- Since String is immutable in java, whenever we do String manipulation like concat, substring etc, it generates a new String and discard the older String for garbage collection.
- These are heavy operations and generate a lot of garbage in heap. So Java has provided StringBuffer and StringBuilder class that should be used for String manipulation.
- StringBuffer and StringBuilder are mutable objects in java and provide append(), insert(), delete() and substring() methods for String manipulation.

1.3. Difference between StringBuffer and StringBuilder

- StringBuffer was the only choice for String manipulation till Java 1.4 but it has one disadvantage that all of its public methods are synchronized. StringBuffer provides Thread safety but on a performance cost.
- In most of the scenarios, we don’t use String in multithreaded environment, so Java 1.5 introduced a new class StringBuilder that is similar with StringBuffer except thread safety and synchronization.

1.4. String vs StringBuffer vs StringBuilder

- String is immutable whereas StringBuffer and StringBuider are mutable classes.
- StringBuffer is thread safe and synchronized whereas StringBuilder is not, that’s why StringBuilder is faster than StringBuffer.
- String concat + operator internally uses StringBuffer or StringBuilder class.

- For String manipulations in non-multi-threaded environment, we should use StringBuilder else use StringBuffer class.

2. Why String class is final or immutable?

- Immutable objects are thread-safe. Two threads can both work on an immutable object at the same time without any possibility of conflict.
- Security: the system can pass on sensitive bits of read-only information without worrying that it will be altered
- You can share duplicates by pointing them to a single instance.
- You can create substrings without copying. You just create a pointer into an existing base String guaranteed never to change. Immutability is the secret that makes Java substring implementation very fast.
- Immutable objects are good fit for becoming Hashtable keys. If you change the value of any object that is used as a hash table key without removing it and re-adding it you will lose the object mapping.
- Since String is immutable, inside each String is a char[] exactly the correct length. Unlike a StringBuilder there is no need for padding to allow for growth.
- If String were not final, you could create a subclass and have two strings that look alike when "seen as Strings", but that are actually different.

3. Is Java Pass by Reference or Pass by Value?

3.1 Pass-by-value

The actual parameter (or argument expression) is fully evaluated and the resulting value is copied into a location being used to hold the formal parameter's value during method/function execution. That location is typically a chunk of memory on the runtime stack for the application (which is how Java handles it), but other languages could choose parameter storage differently.

3.2 Pass-by-reference

The formal parameter merely acts as an alias for the actual parameter. Anytime the method/function uses the formal parameter (for reading or writing), it is actually using the actual parameter.

Java is strictly pass-by-value, exactly as in C. Read the Java Language Specification (JLS). In short: Java has pointers and is strictly pass-by-value. There's no funky rules. It's simple, clean, and clear.

The Java Spec says that everything in java is pass-by-value. There is no such thing as "pass-by-reference" in java.

The key to understanding this is that something like

```
Dog myDog;
```

is not a Dog; it's actually a pointer to a Dog.

What that means, is when you have

```
Dog myDog = new Dog("Rover");
```

```
foo(myDog);
```
you're essentially passing the address of the created Dog object to the foo method. (I say essentially b/c java pointers aren't direct addresses, but it's easiest to think of them that way)

Suppose the Dog object resides at memory address 42. This means we pass 42 to the method.

If the Method were defined as

```
public void foo(Dog someDog) {
    someDog.setName("Max");     // AAA
    someDog = new Dog("Fifi");  // BBB
    someDog.setName("Rowlf");   // CCC
}
```
Let's look at what's happening.

the parameter someDog is set to the value 42

at line "AAA"

someDog is followed to the Dog it points to (the Dog object at address 42) that Dog (the one at address 42) is asked to change his name to Max

at line "BBB"

a new Dog is created. Let's say he's at address 74 we assign the parameter someDog to 74

at line "CCC"

someDog is followed to the Dog it points to (the Dog object at address 74) that Dog (the one at address 74) is asked to change his name to Rowlf then, we return

Now let's think about what happens outside the method:

Did myDog change?

There's the key.Keeping in mind that myDog is a pointer, and not an actual Dog, the answer is NO. myDog still has the value 42; it's still pointing to the original Dog.

4. What is Marker interface? How is it used in Java?

The marker interface is a design pattern, used with languages that provide run-time type information about objects. It provides a way to associate metadata with a class where the language does not have explicit support for such metadata. To use this pattern, a class implements a marker interface, and code that interact with instances of that class test for the existence of the interface. Whereas a typical interface specifies methods that an implementing class must support, a marker interface does not do so. The mere presence of such an interface indicates specific behavior on the part of the implementing class. There can be some hybrid interfaces, which both act as markers and specify required methods, are possible but may prove confusing if improperly used. Java utilizes this pattern very well and the example interfaces are

- java.io.Serializable - Serializability of a class is enabled by the class implementing the java.io.Serializable interface. The Java Classes that do not implement Serializable interface will not be able to serialize or deserializ their state. All subtypes of a serializable class are themselves

serializable. The serialization interface has no methods or fields and serves only to identify the semantics of being serializable.

- java.rmi.Remote - The Remote interface serves to identify interfaces whose methods may be invoked from a non-local virtual machine. Any object that is a remote object must directly or indirectly implement this interface. Only those methods specified in a "remote interface", an interface that extends java.rmi.Remote are available remotely.
- java.lang.Cloneable - A class implements the Cloneable interface to indicate to the Object.clone() method that it is legal for that method to make a field-for-field copy of instances of that class. Invoking Object's clone method on an instance that does not implement the Cloneable interface results in the exception CloneNotSupportedException being thrown.
- javax.servlet.SingleThreadModel - Ensures that servlets handle only one request at a time. This interface has no methods.
- java.util.EvenListener - A tagging interface that all event listener interfaces must extend.

The "instanceof" keyword in java can be used to test if an object is of a specified type. So this keyword in combination with Marker interface can be used to take different actions based on type of interface an object implements.

5. Why main() in java is declared as public static void main? What if the main method is declared as private?

Public - main method is called by JVM to run the method which is outside the scope of project therefore the access specifier has to be public to permit call from anywhere outside the application static - When the JVM makes are call to the main method there is not object existing for the class being called therefore it has to have static method to allow invocation from class. void - Java is platform independent language therefore if it will return some value then the value may mean different to different platforms so unlike C it can not assume a behavior of returning value to the operating system. If main method is declared as private then - Program will compile properly but at run-time it will give "Main method not public." error.

6. Can you write a method to calculate the Nth Fibonacci number?

In mathematics, the Fibonacci numbers or Fibonacci sequence are the numbers in the following integer sequence:

0,1,1,2,3,5,8,13,21,34.

The Fibonacci spiral: an approximation of the golden spiral created by drawing circular arcs connecting the opposite corners of squares in the Fibonacci tiling;this one uses squares of sizes 1, 1, 2, 3, 5, 8, 13, 21, and 34. By definition, the first two numbers in the Fibonacci sequence are 1 and 1, or 0 and 1,

depending on the chosen starting point of the sequence, and each subsequent number is the sum of the previous two.

```
public class FibonacciNumbers {

  /**
   * Recursive solution based on Fibonacci sequence definition. F(N) = F(N-1)
+ F(N-2).
```

```
   *
   * The complexity order of this algorithm is O(2^N) where N is integer used
as parameter. In space
   * terms, the complexity order of this algorithm is O(1) because we are not
using any auxiliary
   * data structure to solve this problem.
   */
  public int getRecursive(int n) {
    validateInput(n);
    if (n == 0 || n == 1) {
      return 1;
    } else {
      return getRecursive(n - 1) + getRecursive(n - 2);
    }
  }

  /**
   * Iterative approach. The complexity order in this algorithm is
O(N) where N is the integer used
   * as parameter. In space terms, the complexity order of this
algorithm is again O(1).
   */
  public int getIterative(int n) {
    validateInput(n);

    if (n <= 1) {
      return 1;
    }
    int previous = 1;
    int current = 1;
    int element = 0;
    for (int i = 2; i <= n; i++) {
      element = previous + current;
      previous = current;
      current = element;
    }
    return element;
  }

  private static int[] elements = new int[1000];

  /**
   * This version of the recursive algorithm is better in performance
terms because we are caching
   * every calculated element to avoid every branch of the recursion
the same values. Is faster
   * because one branch is going to take the already calculated value
from other branches and when
   * you are going to calculate the 11th value the only thing you have
to calculate is to take
   * previous values from the array instead of iterate from nth to 1
and sum every value. I've used
   * a dynamic programming technique.
   *
```

```
   * The problem with this algorithm is related with the space
complexity which is much bigger than
   * the one used for the previous algorithms. In this case, we have
O(N) because we are using an
   * additional data structure to store partial results.
   */
  public int getRecursiveWithCaching(int n) {
    validateInput(n);

    if (n <= 1) {
      return 1;
    } else if (elements[n] != 0) {
      return elements[n];
    }
    elements[n] = getRecursiveWithCaching(n - 1) +
getRecursiveWithCaching(n - 2);
    return elements[n];
  }

  private void validateInput(int n) {
    if (n < 0) {
      throw new IllegalArgumentException("You can't use negative
values as parameter.");
    }
  }
}
```

7. Can you write a method to calculate the factorial of a given positive integer?

```
/**
 * In mathematics, the factorial of a non-negative integer n, denoted
by n!, is the product of all
 * positive integers less than or equal to n. For example:
 *
 * 5! = 5 times 4 times 3 times 2 times  1 = 120.
 *
 * The value of 0! is 1, according to the convention for an empty
product.
 *
 * Can you write a method to calculate the factorial of a given
positive integer?
 *
 */
public class Factorial {

  /**
   * Iterative solution for this problem. This version is based on the
factorial definition
   * described in the statement and the complexity order in time terms of
this solution is O(N)
   * where N is integer passed as argument. In space terms, the complexity
order of this algorithm
```

```
   * is O(1) because we are not using any additional data structure related
to the integer passed
   * as parameter.
   */
  public int getIterative(int n) {
    validateInput(n);

    int result = 1;
    for (int i = n; i >= 2; i--) {
      result *= i;
    }
    return result;
  }

  /**
   * Recursive implementation of the previous algorithm. The
complexity order in time and space
   * terms is the same. Take into account that this implementation is
not tail recursive.
   */
  public int getRecursive(int n) {
    validateInput(n);

    if (n == 0) {
      return 1;
    } else {
      return n * getRecursive(n - 1);
    }
  }

  /**
   * Tail recursive implementation of the previous algorithm. The
complexity order in time and
   * space terms is the same but the resources needed by the CPU to
execute this method is lower
   * because this implementation is tail recursive.
   */
  public int getTailRecursive(int n) {
    validateInput(n);
    return getTailRecursiveInner(n, 1);
  }

  private int getTailRecursiveInner(int n, int acc) {
    if (n == 0) {
      return acc;
    } else {
      return getTailRecursiveInner(n - 1, acc * n);
    }
  }
```

8. Write a Java program to check if a number is Prime or not?

A number is said prime if it is not divisible by any other number except itself. 1 is not considered prime, so your check must start with 2. Simplest solution of this to check every number until the number itself to see if its divisible or not.

8.1 Solution 1

```
public class Prime {

    public static void main(String[] args) {
        int n = 1000;
        for (int i = 0; i < 1000; i++) {
            if (prime(i)) {
                System.out.println(i);
            }
        }
    }
    private static boolean prime(int n) {

        if (n%2 == 0){
            return n==2;
        }
        if (n%3 ==0){
            return n==3;
        }
        if (n%5 ==0){
            return n==5;
        }
        for (int i = 7; i*i < n; i++) {
            if (n%i == 0){
                return false;
            }
        }
        return true;
    }

}
```

8.2 Solution 2

```
public class FirstNPrimes {
      static void printFirstNPrimes(int n) {
             ArrayList<Integer> primes = new ArrayList<Integer>();
             primes.add(2);
             primes.add(3);
             primes.add(5);
             primes.add(7);
             boolean isPrime;
```

```
                for(int i = 8 ; n > primes.size() ; i++) {
                        isPrime = true;
                        for(int prime : primes) {
                                if(i % prime == 0) {
                                        isPrime = false;
                                        break;
                                }
                        }
                        if(isPrime)
                                primes.add(i);
                }

                for(int prime : primes)
                        System.out.println(prime);
        }

        public static void main(String[] args) {
                Scanner in = new Scanner(System.in);
                System.out.println("Enter the n value: ");
                int limit = in.nextInt();
                in.close();
                printFirstNPrimes(limit);
        }
}
```

9. Implement a method String reverse(String str) to reverse a String passed as parameter.

```
public class ReverseString {

  /**
   * Iterative algorithm to solve this problem. The complexity order
in time and space terms of
   * this algorithm is O(N) where N is the number of chars in the
input String. In time terms is
   * possible to get O(N) because String class uses a char array
internally and the complexity
   * order of charAt method is O(1).
   */
  public String reverseIterative(String input) {
    validateInput(input);

    StringBuilder stringBuilder = new StringBuilder();
    for (int i = input.length() - 1; i >= 0; i--) {
      stringBuilder.append(input.charAt(i));
    }
    return stringBuilder.toString();
```

```
  }

  public void reverseIterative2(String input) {
            validateInput(input);
            for (String part : input.split(" ")) {
                System.out.print(new
StringBuilder(part).reverse().toString());
                System.out.print(" ");
            }
          }

  /**
   * Tail recursive solution to this problem. This algorithm has the
same complexity order in time
   * and space terms than the previous approach.
   */
  public String reverseRecursive(String input) {
    validateInput(input);
    return reverseRecursiveInner(input, input.length() - 1, new
StringBuilder());
  }

  private String reverseRecursiveInner(String input, int i,
StringBuilder stringBuilder) {
    if (i < 0) {
      return stringBuilder.toString();
    } else {
      stringBuilder.append(input.charAt(i--));
      return reverseRecursiveInner(input, i, stringBuilder);
    }
  }

  private void validateInput(String input) {
    if (input == null) {
      throw new IllegalArgumentException("You can't pass a null String
as input parameter.");
    }
  }
}
```

10. Difference between Association, Composition and Aggregation in Java

In Object-oriented programming, one object is related to other to use functionality and service provided by that object. This relationship between two objects is known as the association in object oriented general software design and depicted by an arrow in Unified Modelling language or UML. Both Composition and Aggregation are the form of association between two objects, but there is a subtle difference between composition and aggregation, which is also reflected by their UML notation. We refer association between two objects as Composition, when one class owns other class and other class cannot meaningfully exist, when it's owner destroyed, for example, Human class is a composition of

several body parts including Hand, Leg and Heart. When human object dies, all its body part ceased to exist meaningfully, this is one example of Composition.

In Object-oriented programming, one object is related to other to use functionality and service provided by that object. This relationship between two objects is known as the association in object oriented general software design and depicted by an arrow in Unified Modelling language or UML. Both Composition and Aggregation are the form of association between two objects, but there is a subtle difference between composition and aggregation, which is also reflected by their UML notation. We refer association between two objects as Composition, when one class owns other class and other class can not meaningfully exist, when it's owner destroyed, for example, Human class is a composition of several body parts including Hand, Leg and Heart. When human object dies, all it's body part ceased to exist meaningfully, this is one example of Composition.

Another example of Composition is Car and it's part e.g. engines, wheels etc. Individual parts of the car can not function when a car is destroyed. While in the case of Aggregation, including object can exists without being part of the main object e.g. a Player which is part of a Team, can exist without a team and can become part of other teams as well.

Another example of Aggregation is Student in School class, when School closed, Student still exist and then can join another School or so. In UML notation, a composition is denoted by a filled diamond, while aggregation is denoted by an empty diamond, which shows their obvious difference in terms of strength of the relationship.

The composition is stronger than Aggregation. In Short, a relationship between two objects is referred as an association, and an association is known as composition when one object owns other while an association is known as aggregation when one object uses another object.

10.1 Composition

Since Engine is-part-of Car, the relationship between them is Composition. Here is how they are implemented between Java classes.

```
public class Car {
    //final will make sure engine is initialized
    private final Engine engine;

    public Car(){
       engine  = new Engine();
    }
}
class Engine {
    private String type;
}
```

10.2 Aggregation

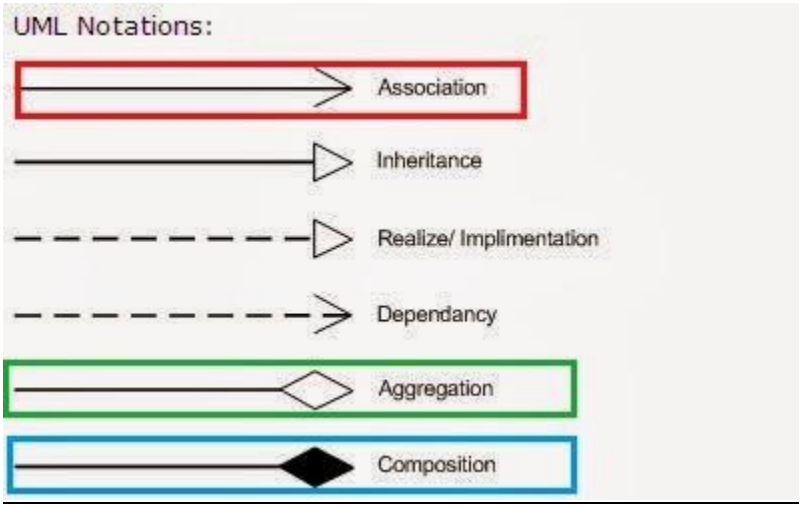
Since Organization has Person as employees, the relationship between them is Aggregation. Here is how they look like in terms of Java classes

```
public class Organization {
```

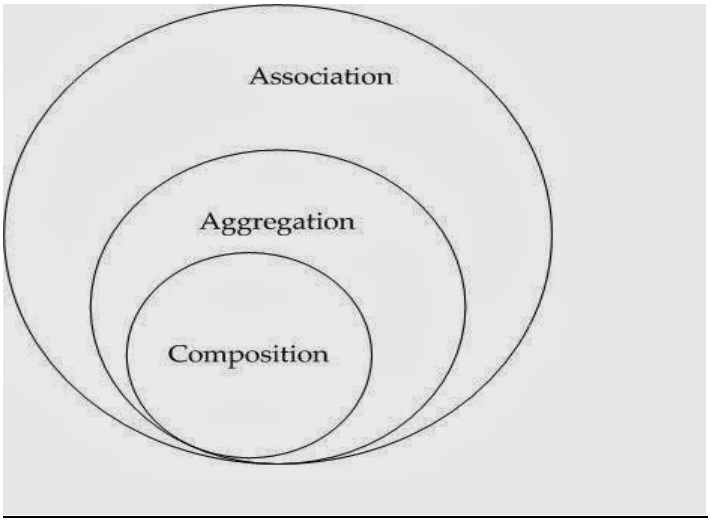
```
    private List employees;
}

public class Person {
    private String name;
}
```

10.3 UML Diagram of Association, Composition and Aggregation

UML has different notations to denote aggregation, composition and association. Association is denoted by the simple arrow while aggregation is denoted by empty diamond-head arrow and composition is denoted by filled diamond-head arrow. When you draw UML diagram for two related class A and B, where A is associated with B then its denoted by A -> B. Similar way is used to show aggregation and composition between two classes. Here are UML notations for different kind of dependency between two classes.



As I said all three denotes relationship between object and only differ in their strength, you can also view them as below, where composition represents strongest form of relationship and association being the most general form.



10.4 Association vs Composition vs Aggregation

Here is the list of differences between Composition and Aggregation in point format, for quick review. As I said the key difference between them comes from the point that in the case of Composition, One object is OWNER of another object, while in the case of aggregation, one object is just a USER or another object.

1. If A and B two classes are related to each other such that, B ceased to exist, when A is destroyed, then the association between two objects is known as Composition. An example is Car and Engine. While if A and B are associated with each other, such that B can exist without being associated with A, then this association in known as Aggregation.

2. In the case of Composition A owns B e.g. Person is the owner of his Hand, Mind and Heart, while in the case of Aggregation, A uses B e.g. Organization uses People as an employee.

3. In UML diagram Association is denoted by a normal arrow head, while Composition is represented by filled diamond arrow head, and Aggregation is represented by an empty diamond arrow head, As shown in below and attached diagram in the third paragraph.

```
Association A---->B
Composition A-----<filled>B
Aggregation A-----<>B
```

4. Aggregation is a lighter form of Composition, where a sub-part object can meaningfully exist without main objects.

5. In Java, you can use final keyword to represent Composition. Since in Composition, Owner object expects a part object to be available and functions, by making it final, your provide guarantee that, when Owner will be created, this part object will exist. This is actually a Java idiom to represent a strong form of association i.e. composition between two objects.

6. Another interesting word, which comes handy to understand difference between Composition and Aggregation in software design is "part-of" and "has". If one object is-part-of another object e.g. Engine is part of Car, then association or relationship between them is Composition. On the other hand, if one object just has another object e.g. Car has the driver then it's Aggregation.

11. Difference between Abstract class vs Interface

1) Interface in Java can only contains declaration. You can not declare any concrete methods inside interface. On the other hand abstract class may contain both abstract and concrete methods, which makes abstract class an ideal place to provide common or default functionality. I suggest reading my post 10 things to know about interface in Java to know more about interfaces, particularly in Java programming language.

2) Java interface can extend multiple interface also Java class can implement multiple interfaces, Which means interface can provide more Polymorphism support than abstract class . By extending abstract class, a class can only participate in one Type hierarchy but by using interface it can be part of multiple type hierarchies. E.g. a class can be Runnable and Displayable at same time. One example I can remember of this is writing GUI application in J2ME, where class extends Canvas and implements CommandListener to provide both graphic and event-handling functionality.

3) In order to implement interface in Java, until your class is abstract, you need to provide implementation of all methods, which is very painful. On the other hand abstract class may help you in this case by providing default implementation. Because of this reason, I prefer to have minimum methods in interface, starting from just one, I don't like idea of marker interface, once annotation is introduced in Java 5. If you look JDK or any framework like Spring, which I does to understand OOPS and design patter better, you will find that most of interface contains only one or two methods e.g. Runnable, Callable, ActionListener etc.

11.1 When to use interface and abstract class in Java

1) In Java particularly, decision between choosing Abstract class and interface may influence by the fact that multiple inheritance is not supported in Java. One class can only extend another class in Java. If you choose abstract class over interface than you lost your chance to extend another class, while at the same time you can implement multiple interfaces to show that you have multiple capability. One of the common example, in favor of interface over abstract class is Thread vs Runnable case. If you want to execute a task and need run() method it's better to implement Runnable interface than extending Thread class.

2) Let's see another case where an abstract class suits better than interface. Since abstract class can include concrete methods, it’s great for maintenance point of view, particularly when your base class is evolving and keep changing. If you need a functionality across all your implementation e.g. a common method, than, you need to change every single implementation to include that change if you have chosen interface to describe your base class. Abstract class comes handy in this case because you can just define new functionality in abstract super class and every sub class will automatically gets it. In short, abstract class are great in terms of evolving functionality. If you are using interface, you need to exercise extra care while defining contracts because it’s not easy to change them once published.

3) Interface in Java is great for defining Types. Programming for interfaces than implementation is also one of the useful Object oriented design principle which suggests benefit of using interface as argument to function, return type etc.

4) One more general rule of when to use abstract class and interface is to find out whether a certain class will form a IS-A hierarchy or CAN-DO-THIS hierarchy. If you know that you will be creating classes e.g. Circle, Square than it's better to create an abstract class Shape which can have area() and

perimeter() as abstract method, rather than defining Shape as interface in Java. On the other hand if you are going to create classes which can do things like, can fly, you can use interface Flyable instead of abstract class.

5) Interface generally define capability e.g. Runnable can run(), Callable can call(), Displayable can display(). So if you need to define capability, consider using interface. Since a class can have multiple capabilities i.e. a class can be Runnable as well as Displayable at same time. As discussed in first point, since java does not allow multiple inheritance at class level, only way to provide multiple capability is via interfaces.

6) Let's see another example of where to use Abstract class and Interface in Java, which is related to earlier point. Suppose you have lot of classes to model which are birds, which can fly, than creating a base abstract class as Bird would be appropriate but if you have to model other things along with Birds, which can fly e.g. Airplanes, Balloons or Kites than it's better to create interface Flyable to represent flying functionality. In conclusion, if you need to provide a functionality which is used by same type of class than use Abstract class and if functionality can be used by completely unrelated classes than use interface.

7) Another interesting use of Abstract class and interface is defining contract using interface and providing skeletal using abstract class. java.util.List from Java collection framework is a good example of this pattern. List is declared as interface and extends Collection and Iterable interface and AbstractList is an abstract class which implements List. AbstractList provides skeletal implementation of List interface. Benefit of using this approach is that it minimize the effort to implement this interface by concrete class e.g. ArrayList or LinkedList. If you don't use skeletal implementation e.g. abstract class and instead decide to implement List interface than not only you need to implement all List methods but also you might be duplicating common code. Abstract class in this case reduce effort to implement interface.

8) Interface also provide more decoupling than abstract class because interface doesn't contain any implementation detail, while abstract class may contain default implementation which may couple them with other class or resource.

9) Using interface also help while implementing Dependency Injection design pattern and makes testing easy. Many mock testing framework utilize this behavior.

10). From Java 8 onwards, we can have method implementations in the interfaces. We can create default as well as static methods in the interfaces and provide implementation for them. This has bridge the gap between abstract classes and interfaces and now interfaces are the way to go because we can extend it further by providing default implementations for new methods.

12. What is Abstraction?

Abstraction is the concept of exposing only the required essential characteristics and behavior with respect to a context.

12.1 Abstraction Layers

When we see a nice car on the road as a casual onlooker, we get to see the whole picture. The car as a one single unit, a vehicle. We do not see the underlying complex mechanical engineering.

Now consider we are going to a showroom to buy a car. What do we see now? We see four wheels, powerful engine, power steering etc. We see the car at high level components. But, there is so much inside it which gives the completeness to the car.

Now consider a mechanic, who is going to service the car. He will see one more level deeper with more level of information.

When we design software, we take the context. In the above example, we ask the question whether we are designing the software for a causal on looker or a buyer or a mechanic? Levels of abstraction is applied on the design accordingly.

As per dictionary, abstraction is the quality of dealing with ideas rather than events. For example, when you consider the case of e-mail, complex details such as what happens as soon as you send an e-mail, the protocol your e-mail server uses are hidden from the user. Therefore, to send an e-mail you just need to type the content, mention the address of the receiver, and click send.

Likewise in Object-oriented programming, abstraction is a process of hiding the implementation details from the user, only the functionality will be provided to the user. In other words, the user will have the information on what the object does instead of how it does it.

In Java, abstraction is achieved using Abstract classes and interfaces.

12.2 Abstraction in OOP

In general computer software, when we talk about abstraction the software language itself is an example for the concept of abstraction. When we write a statement as,

```
a = b + c;
```

We are adding two values stored in two locations and then storing the result in a new location. We just describe it in an easily human understandable form. What happens beneath? There are registers, instruction sets, program counters, storage units, etc involved. There is PUSH, POP happening. High level language we use abstracts those complex details.

When we say abstraction in Java, we are talking about abstraction in object oriented programming (OOP) and how it is done in Java. Concept of abstraction in OOP, starts right at the moment when a class is getting conceived. I will not say, using Java access modifiers to restrict the properties of an object alone is abstraction. There is lot more to it. Abstraction is applied everywhere in software and OOP.

12.3 Abstraction and Encapsulation

When a class is conceptualized, what are the properties we can have in it given the context. If we are designing a class Animal in the context of a zoo, it is important that we have an attribute as animalType to describe domestic or wild. This attribute may not make sense when we design the class in a different context.

Similarly, what are the behaviors we are going to have in the class? Abstraction is also applied here. What is necessary to have here and what will be an overdose? Then we cut off some information from the class. This process is applying abstraction.

When we ask for difference between encapsulation and abstraction, I would say, encapsulation uses abstraction as a concept. So then, is it only encapsulation. No, abstraction is even a concept applied as part of inheritance and polymorphism.

We got to look at abstraction at a level higher among the other OOP concepts encapsulation, inheritance and polymorphism.

12.4 Abstraction and Inheritance

Let us take inheritance also in this discussion. When we design the hierarchy of classes, we apply abstraction and create multiple layers between each hierarchy. For example, lets have a first level class Cell, next level be LivingBeing and next level be Animal. The hierarchy we create like this based on the context for which we are programming is itself uses abstraction. Then for each levels what are the properties and behaviors we are going to have, again abstraction plays an important role here in deciding that.

What are some common properties that can be exposed and elevated to a higher level, so that lower level classes can inherit it. Some properties need not be kept at higher level. These decision making process is nothing but applying abstraction to come up with different layers of hierarchy. So abstraction is one key aspect in OOP as a concept.

Abstraction and encapsulation are complementary concepts: abstraction focuses on the observable behavior of an object... encapsulation focuses upon the implementation that gives rise to this behavior... encapsulation is most often achieved through information hiding, which is the process of hiding all of the secrets of object that do not contribute to its essential characteristics.

In other words: abstraction = the object externally; encapsulation (achieved through information hiding) = the object internally.

12.5 Abstraction in Java

Having read the above section, you might have now come to an idea of how abstraction is done in Java.

- When we conceptualize a class
- When we write an ‘interface’
- When we write an ‘abstract’ class, method
- When we write ‘extends’
- When we apply modifiers like ‘private’.

12.6 What is abstract class

An abstract class is something which is incomplete and you cannot create an instance of the abstract class. If you want to use it you need to make it complete or concrete by extending it. A class is called concrete if it does not contain any abstract method and implements all abstract method inherited from abstract class or interface it has implemented or extended. By the way Java has a concept of abstract classes, abstract method but a variable cannot be abstract in Java.

An abstract method in Java doesn't have the body, it's just a declaration. In order to use an abstract method, you need to override that method in sub class.

12.7 So when do you use abstraction?

When you know something needs to be there but not sure how exactly it should look like. e.g. when I am creating a class called Vehicle, I know there should be methods like start() and stop() but don't know how that start and stop method should work, because every vehicle can have different start and stop mechanism e.g. some can be started by kicking or some can be by pressing buttons . The Same concept applies to interface in Java as well, which we will discuss in some other post.

So the implementation of those start() and stop() methods should be left to their concrete implementation e.g. Scooter, MotorBike , Car etc.

Abstraction Using Interface:

In Java Interface is an another way of providing abstraction, Interfaces are by default abstract and only contains public, static, final constant or abstract methods. It's very common interview question is that where should we use abstract class and where should we use Java Interfaces in my view this is important to understand to design better Java application, you can go for java interface if you only know the name of methods your class should have e.g. for Server it should have start() and stop() method but we don't know how exactly these start and stop method will work.

If you know some of the behavior while designing class and that would remain common across all subclasses add that into an abstract class. An interface like Runnable interface is a good example of abstraction in Java which is used to abstract task executed by multiple threads. Callable is another good abstract of a task which can return value.

12.8 Abstraction: Things to Remember

1) Use abstraction if you know something needs to be in class but the implementation of that varies. Abstraction is actually resulting of thought process and it really need good experience of both domain and Object oriented analysis and design to come up with good abstraction for your project.

2) In Java, you cannot create an instance of the abstract class using the new operator, its compiler error. Though abstract class can have a constructor.

3) Abstract is a keyword in Java, which can be used with both class and method. Abstract class can contain both abstract and concrete method. An abstract method doesn't have the body, just declaration.

4) A class automatically becomes abstract class when any of its methods declared as abstract.

5) Abstract method doesn't have method body.

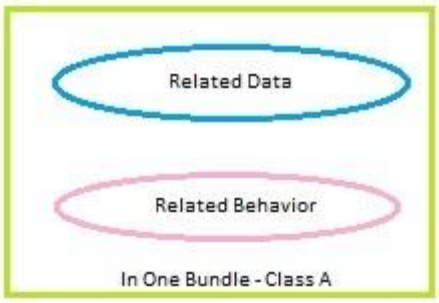
6) In Java, a variable cannot be made abstract, its only class or methods which would be abstract.

7) If a class extends an abstract class or interface it has to provide implementation to all its abstract method to be a concrete class. Alternatively, this class can also be abstract.

13. What is Encapsulation?

Encapsulation is the ability to package data, related behavior in an object bundle and control/restrict access to them (both data and function) from other objects. It is all about packaging related stuff together and hide them from external elements.

We can see that keywords encapsulation and data hiding are used synonymously everywhere. It should not be misunderstood that encapsulation is all about data hiding only. When we say encapsulation, emphasis should be on grouping or packaging or bundling related data and behavior together.



When we design a class in OOP, the first principle we should have in mind is encapsulation. Group the related data and its behavior in a bucket. Primary benefit of encapsulation is, better maintainability.

That's it! This is all about encapsulation. Let us leave it as simple as that. Nothing more and nothing less.

13.1 Encapsulation in Java

Any well-defined Java class in its context domain is an example for encapsulation in Java. So the importance here is for bundling data and methods together only after that data hiding comes into picture. When we say data hiding, we all know about access specifiers in Java. Hiding an attribute or method is easier from the outer world. Just use the access specifier 'private'.

So data hiding is fairly straight forward. How about bundling data and method. This is critical. We should understand well about the business domain and based on that design the class and group attributes and its methods together. This process is key in encapsulation.

Following is an outline example for Java encapsulation. When we talk about an animal, we should have all its attributes listed, similarly its behavior like how it will hunt, run, mate, etc. By bundling all these data and behavior in a single class we are following encapsulation principles.

```
package com.javapapers.java;

public class Animal {
	private String animalName;
	private String animalType;
	private int height;
	private String color;

	public Animal(String animalName, String animalType) {
		this.animalName = animalName;
		this.animalType = animalType;
```

```
        }

        public void hunt() {
                // implementation of hunt
        }

        public void run() {
                // implementation of run
        }

        public void mate() {
                // implementation of mate
        }
        //encapsulation is not about having getter/setters
        public String getAnimalName() {

                return animalName;
        }

        public void setAnimalName(String animalName) {
                this.animalName = animalName;
        }

        public String getAnimalType() {
                return animalType;
        }

        public void setAnimalType(String animalType) {
                this.animalType = animalType;
        }

}
```

Encapsulation in Java or object oriented programming language is a concept which enforces protecting variables, functions from outside of class, in order to better manage that piece of code and having least impact or no impact on other parts of a program due to change in protected code. Encapsulation in Java is visible at different places and Java language itself provide many constructs to encapsulate members. You can completely encapsulate a member be it a variable or method in Java by using private keyword and you can even achieve a lesser degree of encapsulation in Java by using other access modifiers like protected or the public.

Encapsulation is nothing but protecting anything which is prone to change. Rational behind encapsulation is that if any functionality which is well encapsulated in code i.e. maintained in just one place and not scattered around code is easy to change. this can be better explained with a simple example of encapsulation in Java. we all know that constructor is used to creating object in Java and constructor can accept an argument. Suppose we have a class Loan has a constructor and then in various classes, you have created an instance of the loan by using this constructor. now requirements change and you need to include the age of borrower as well while taking a loan.

Since this code is not well encapsulated i.e. not confined in one place you need to change everywhere you are calling this constructor i.e. for one change you need to modify several file instead of just one file which is more error prone and tedious, though it can be done with refactoring feature of advanced IDE wouldn't it be better if you only need to make change at one place? Yes, that is possible if we

encapsulate Loan creation logic in one method say createLoan() and client code call this method and this method internally create Loan object. in this case, you only need to modify this method instead of all client code.

13.2 Example of Encapsulation

```
class Loan{
    private int duration;  //private variables examples of
encapsulation
    private String loan;
    private String borrower;
    private String salary;

    //public constructor can break encapsulation instead use factory
method
    private Loan(int duration, String loan, String borrower, String
salary){
        this.duration = duration;
        this.loan = loan;
        this.borrower = borrower;
        this.salary = salary;
    }

    //no argument constructor omitted here

   // create loan can encapsulate loan creation logic
    public Loan createLoan(String loanType){

     //processing based on loan type and then returning loan object
      return loan;
    }
}
```

In this same example of Encapsulation in Java, you see all member variables are made private so they are well encapsulated you can only change or access this variable directly inside this class. if you want to allow outside world to access these variables is better creating a getter and setter e.g. getLoan() that allows you to do any kind of validation, security check before return loan so it gives you complete control of whatever you want to do and single channel of access for client which is controlled and managed.

13.3 Advantage of Encapsulation

1. Encapsulated Code is more flexible and easy to change with new requirements.
2. Encapsulation in Java makes unit testing easy.
3. Encapsulation in Java allows you to control who can access what.
4. Encapsulation also helps to write immutable class in Java which is a good choice in multi-threading environment
5. Encapsulation reduces coupling of modules and increases cohesion inside a module because all piece of one thing is encapsulated in one place.
6. Encapsulation allows you to change one part of code without affecting other parts of code.

13.4 Design Pattern based on Encapsulation

Many design pattern in Java uses encapsulation concept, one of them is Factory pattern which is used to create objects. Factory pattern is a better choice than new operator for creating an object of those classes whose creation logic can vary and also for creating different implementation of the same interface. BorderFactory class of JDK is a good example of encapsulation in Java which creates different types of Border and encapsulate creation logic of Border. Singleton pattern in Java also encapsulate how you create an instance by providing getInstance() method. since an object is created inside one class and not from any other place in code you can easily change how you create an object without affect another part of code.

13.5 Important points about encapsulation

1. "Whatever changes encapsulate it" is a famous design principle.
2. Encapsulation helps in loose coupling and high cohesion of code.
3. Encapsulation in Java is achieved using access modifier private, protected and public.
4. Factory pattern, Singleton pattern in Java makes good use of Encapsulation.

13.6 Abstraction vs Encapsulation

1. First difference between Abstraction and Encapsulation is that, Abstraction is implemented in Java using interface and abstract class while Encapsulation is implemented using private, package-private and protected access modifier.
2. Encapsulation is also called data hiding.
3. Design principles "programming for interface than implementation" is based on abstraction and "encapsulate whatever changes" is based upon Encapsulation.

14. What is Polymorphism?

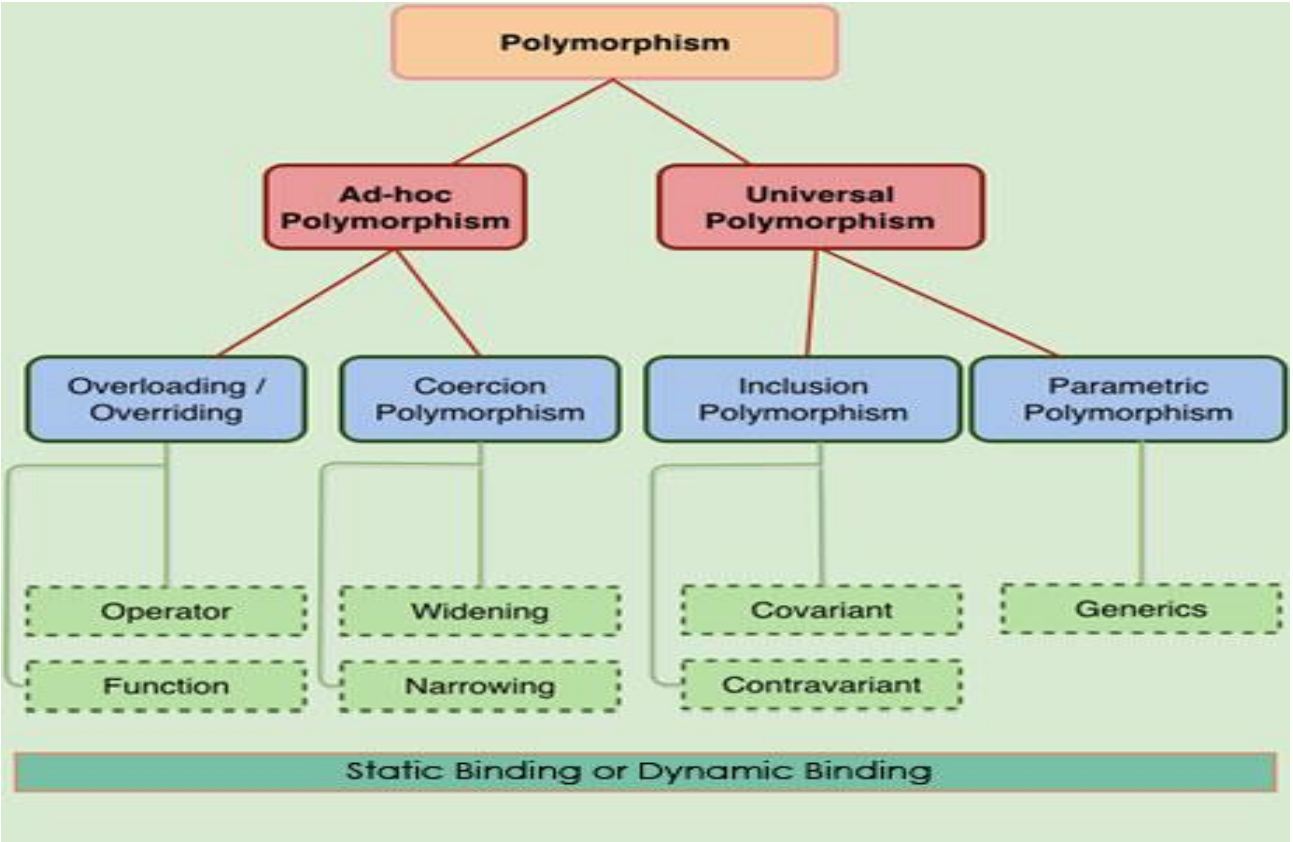
A simplest definition in computer terms would be, handling different data types using the same interface. In this tutorial, we will learn about what is polymorphism in computer science and how polymorphism can be used in Java.

14.1 Types of Polymorphism

Polymorphism in computer science was introduced in 1967 by Christopher Strachey. Please let me know with reference if it is not a fact and the tutorial can be updated. Following are the two major types of polymorphism as defined by Strachey.

1. Ad hoc Polymorphism
2. Parametric Polymorphism

Later these were further categorized as below:



14.1.1 Ad hoc Polymorphism

"Ad-hoc polymorphism is obtained when a function works, or appears to work, on several different types (which may not exhibit a common structure) and may behave in unrelated ways for each type. Parametric polymorphism is obtained when a function works uniformly on a range of types; these types normally exhibit some common structure." – Strachey 1967

If we want to say the above paragraph in two words, they are operator overloading and function overloading. Determining the operation of a function based on the arguments passed.

Ad hoc Polymorphism in Java:

In Java we have function overloading and we do not have operator overloading. Yes we have "+" operator implemented in a polymorphic way.

```
String fruits = "Apple" + "Orange";
int a = b + c;
```

The definition is when the type is different, the internal function adjusts itself accordingly. int and float are different types and so even the following can be included in polymorphism operator overloading.

```
int i = 10 - 3;
float f = 10.5 - 3.5;
```

Similarly even * and / can be considered as overloaded for int and float types.

Having said all the above, these are all language implemented features. Developers cannot custom overload an operator. So answer for the question, "does Java supports operator overloading?" is "yes and no".

Java wholeheartedly supports function overloading. We can have same function name with different argument type list. For function overloading in Java I have already written a super-hit tutorial and I am sure you will enjoy reading it.

In inheritance, the ability to replace an inherited method in the subclass by providing a different implementation is overriding. Function overriding in is discussed in the same tutorial as overloading.

Polymorphism is a larger concept which consists of all these different types. So it is not right to say that overloading or overriding alone is polymorphism. It is more than that.

14.1.1.1 Coercion Polymorphism

Implicit type conversion is called coercion polymorphism. Assume that we have a function with argument int. If we call that function by passing a float value and if the the run-time is able to convert the type and use it accordingly then it is coercion polymorphism.

Now with this definition, let us see if Java has coercion polymorphism. The answer is half yes. Java supports widening type conversion and not narrowing conversions.

1. Narrowing Conversion:

```
class FToC {
    public static float fToC (int fahrenheit) {
        return (fahrenheit - 32)*5/9;
    }

    public static void main(String args[]) {
        System.out.println(fToC(98.4));
    }
}
```

Java does not support narrowing conversion and we will get error as "FToC.java:7: fToC(int) in FToC cannot be applied to (double)"

2. Widening Conversion:

```
class FToC {
    public static float fToC (float fahrenheit) {
        return (fahrenheit - 32)*5/9;
    }

    public static void main(String args[]) {
        System.out.println(fToC(98));
    }
}
```

The above code will work without an error in Java. We are passing an int value ’98’ wherein the expected value type is a float. Java implicitly converts int value to float and it supports widening conversion.

14.1.2 Universal Polymorphism

Universal polymorphism is the ability to handle types universally. There will be a common template structure available for operations definition irrespective of the types. Universal polymorphism is categorized into inclusion polymorphism and parametric polymorphism.

14.1.2.1 Inclusion polymorphism (subtype polymorphism)

Substitutability was introduced by eminent Barbara Liskov and Jeannette Wing. It is also called as Liskov substitution principle.

"Let T be a super type and S be its subtype (parent and child class). Then, instances (objects) of T can be substituted with instances of S."

Replacing the supertype's instance with a subtype's instance. This is called inclusion polymorphism or subtype polymorphism. This is covariant type and the reverse of it is contravariant. We have discussed the substitution principle and covariant types, contravariant and invariant earlier in the linked tutorial. This is demonstrated with a code example. Java supports subtype polymorphism from Java / JDK version 1.5.

14.1.2.2 Parametric Polymorphism

Here we go, we have come to 'Generics'. This is a nice topic and requires a full detailed tutorial with respect to Java. For now, parametric polymorphism is the ability to define functions and types in a generic way so that it works based on the parameter passed at runtime. All this is done without compromising type-safety.

The following source code demonstrates a generics feature of Java. It gives the ability to define a class and parameterize the type involved. The behavior of the class is based on the parameter type passed when it is instantiated.

```
package com.javapapers.java;

import java.util.ArrayList;
import java.util.List;

public class PapersJar {
	private List itemList = new ArrayList();

	public void add(T item) {
		itemList.add(item);
	}

	public T get(int index) {
		return itemList.get(index);
	}

	public static void main(String args[]) {
		PapersJar papersStr = new PapersJar();
		papersStr.add("Lion");
		String str = papersStr.get(0);
		System.out.println(str);
```

```
                PapersJar papersInt = new PapersJar();
                papersInt.add(new Integer(100));
                Integer integerObj = papersInt.get(0);
                System.out.println(integerObj);
        }
}
```

Java started to support parametric polymorphism with introduction of Generic in JDK1.5. Collection classes in JDK 1.5 are written using Generic Type which allows Collections to hold any type of object at run time without any change in code and this has been achieved by passing actual Type as parameter. For example see the below code of a parametric cache written using Generic which shows use of parametric polymorphism in Java. Read how to create Generic class and methods in Java for more details.

```
interface cache{
  public void put(K key, V value);
  public V get(K key);
}
```

14.2 Static Binding vs Dynamic Binding

Give all the above polymorphism types, we can classify these under different two broad groups static binding and dynamic binding. It is based on when the binding is done with the corresponding values. If the references are resolved at compile time, then it is static binding and if the references are resolved at runtime then it is dynamic binding. Static binding and dynamic binding also called as early binding and late binding. Sometimes they are also referred as static polymorphism and dynamic polymorphism.

Let us take overloading and overriding for example to understand static and dynamic binding. In the below code, first call is dynamic binding. Whether to call the obey method of DomesticAnimal or Animal is resolve at runtime and so it is dynamic binding. In the second call, whether the method obey() or obey(String i) should be called is decided at compile time and so this is static binding.

```
package com.javapapers.java;

public class Binding {

        public static void main(String args[]) {
                Animal animal = new DomesticAnimal();
                System.out.println(animal.obey());

                DomesticAnimal domesticAnimal = new DomesticAnimal();
                System.out.println(domesticAnimal.obey("Ok!"));
        }
}

class Animal {
        public String obey() {
                return "No!";
        }

}
```

```
class DomesticAnimal extends Animal {
        public String obey() {
                return "Yes!";
        }

        public String obey(String i) {
                return i;
        }
}
Output:

Yes!

Ok!
```

14.3 Advantages of Polymorphism

1. Generics: Enables generic programming.
2. Extensibility: Extending an already existing system is made simple.
3. De-clutters the object interface and simplifies the class blueprint.

14.4 What is polymorphism in Java?

Polymorphism is an Oops concept which advice use of common interface instead of concrete implementation while writing code. When we program for interface our code is capable of handling any new requirement or enhancement arise in near future due to new implementation of our common interface. If we don't use common interface and rely on concrete implementation, we always need to change and duplicate most of our code to support new implementation. Its not only Java but other object oriented language like C++ also supports polymorphism and it comes as fundamental along with other OOPS concepts like Encapsulation , Abstraction and Inheritance.

14.5 How Polymorphism supported in Java

Java has excellent support of polymorphism in terms of Inheritance, method overloading and method overriding. Method overriding allows Java to invoke method based on a particular object at run-time instead of declared type while coding. To get hold of concept let's see an example of polymorphism in Java:

```
public class TradingSystem{
   public String getDescription(){
      return "electronic trading system";
   }
}

public class DirectMarketAccessSystem extends TradingSystem{
   public String getDescription(){
     return "direct market access system";
   }
}

public class CommodityTradingSystem extends TradingSystem{
```

```
    public String getDescription(){
      return "Futures trading system";
    }
}
```

Here we have a super class called TradingSystem and there two implementation DirectMarketAccessSystem and CommodityTradingSystem and here we will write code which is flexible enough to work with any future implementation of TradingSystem we can achieve this by using Polymorphism in Java which we will see in the further example.

14.6 Where to use Polymorphism in code

Probably this is the most important part of this Java Polymorphism tutorial and It's good to know where you can use Polymorphism in Java while writing code. Its common practice to always replace concrete implementation with interface it's not that easy and comes with practice but here are some common places where I check for polymorphism:

1. Method argument:

Always use super type in method argument that will give you leverage to pass any implementation while invoking a method. For example:

```
public void showDescription(TradingSystem tradingSystem){
    tradingSystem.description();
}
```

If you have used concrete implementation e.g. CommodityTradingSystem or DMATradingSystem then that code will require frequent changes whenever you add new Trading system.

2. Variable names:

Always use Super type while you are storing reference returned from any Factory method in Java, This gives you the flexibility to accommodate any new implementation from Factory. Here is an example of polymorphism while writing Java code which you can use retrieving reference from Factory:

```
String systemName = Configuration.getSystemName();
TradingSystem system = TradingSystemFactory.getSystem(systemName);
```

3. Return type of method

The return type of any method is another place where you should be using interface to take advantage of Polymorphism in Java. In fact, this is a requirement of Factory design pattern in Java to use interface as a return type for factory method.

```
public TradingSystem getSystem(String name){
    //code to return appropriate implementation
}
```

14.7 Method overloading and method overriding in Java

Method overloading and method overriding uses concept of Polymorphism in Java where method name remains same in two classes but actual method called by JVM depends upon object at run time and

done by dynamic binding in Java. Java supports both overloading and overriding of methods. In case of overloading method signature changes while in case of overriding method signature remains same and binding and invocation of method is decided on runtime based on actual object. This facility allows Java programmer to write very flexibly and maintainable code using interfaces without worrying about concrete implementation. One disadvantage of using Polymorphism in code is that while reading code you don't know the actual type which annoys while you are looking to find bugs or trying to debug program. But if you do Java debugging in IDE you will definitely be able to see the actual object and the method call and variable associated with it.

15. What is Inheritance?

Inheritance in Java is an Object oriented or OOPS concepts, which allows to emulate real world Inheritance behavior, Inheritance allows code reuse in Object oriented programming language e.g. Java.

Inheritance in Java is way to define relationship between two classes. Inheritance defines Parent and Child relationship between two classes. Similar to real world where a Child inherit his parents Surname and other qualities, In Java programming language, Child class inherit its Parent's property and code. In Java programming, terms Super class and Sub Class is used to represent Parent and Child class, while in other object oriented language like C++, terms base and derived class is used to denote Inheritance. Inheritance is also simplest but not optimal way to reuse code and When one class use Inheritance to extend another class it get access to all non private code written in Super class. In Context of Inheritance in Java, following are legal :

1) Super class reference variable can point to Sub Class Object e.g.

```
SuperClass parent = new SubClass();
```

is legal at compile time because of IS-A relationship between Super class and Sub Class. In Java Sub Class IS-A Super class like Mango IS-A Fruit.

Similarly you can pass Sub class object in method arguments where Super class is expected, return Sub Class instance where return type of method is Super Class etc.

On the other hand if an you want to store object of Sub class, which is stored in super class reference variable, back on Sub class reference variable you need to use casting in Java, as shown below :

```
SubClass child = (SubClass) parent; //since parent variable pointing
to SubClass object
```

This code will throw ClassCastException if parent reference variable doesn't point to a SubClass object.

15.1 Important points about Inheritance

1. As I said earlier Inheritance in Java is supported using extends and implements keyword, extends keyword is used to inherit from another Java Class and allow to reuse functionality of Parent class. While implements keyword is used to implement Interface in Java. Implementing an interface in Java doesn't actually meant for code reuse but provides Type hierarchy support. You can also use extends keyword when one interface extends another interface in Java.

2. If you do not want to allow Inheritance for your class than you can make it final. final classes can not be extended in Java and any attempt to inherit final class will result in compile time error.

3. Constructor in Java are not inherited by Sub Class. In face Constructors are chained, first statement in constructor is always a call to another constructor, either implicitly or explicitly. If you don't call any other constructor compiler will insert super(), which calls no argument constructor of super class in Java. this keyword represent current instance of class and super keyword represent instance of super class in Java.

4. Inheritance in Java represents IS-A relationship. If you see IS-A relationship between your domain Objects and Classes than consider using Inheritance e.g. if you have a class called ProgrammingLanguage than Java IS-A ProgrammingLanguage and should inherit from ProgrammingLanguage class.

5. Private members of Super class is not visible to Sub class even after using Inheritance in Java. Private members include any private field or method in Java.

6. Java has a special access modifier known as protected which is meant to support Inheritance in Java. Any protected member including protected method and field are only accessible in Child class or Sub class outside the package on which they are declared.

7. One of the risk of Inheritance in Java is that derived class can alter behavior of base class by overriding methods, which can compromise variants of base class e.g. if a malicious class overrides String's equals method in Java to change the comparison logic, you may get different behavior when String reference variable points to that class. to prevent such malicious overriding, you can make your class final to disallow inheritance. But beware making a class final severely implements its client's ability to reuse code. It make more sense from security perspective and that's one of the reason Why String is final in Java.

8. Use @Override annotation while overriding super class's method in subclass. This will ensure a compile time check on whether overriding method actually overrides super class method or not. Its common mistake to overload a method instead of overriding it mostly when super class method accept Object type, common examples are equals method, compareTo method and compare() method in Java.

15.2 When to use Inheritance

Many programmer says favor composition over Inheritance which is true but there are cases where Inheritance is a natural choice, Even in Java API there are many places where inheritances is used e.g. In Java collection framework most of concrete collection classes inherit from there Abstract counterpart e.g. HashSet extends AbstractSet , LinkedHashSet extends HashSet, ArrayList extends AbstractList etc. My general policy to decide whether to use Inheritance or not is to check "IS-A" relationship. For example all above example of Inheritance satisfy IS-A rule e.g. HashSet IS-A Set. Similarly if you have class called Fruit and want to create another class called Mango, its best to use inheritance and Mango should extend Fruit because Mango is a Fruit. By extending Fruit class it gets common state and behavior of Fruit object. Conversely if you find HAS-A relationship between two classes than use Composition e.g. Car HAS-A Seat, So Car class should be composed with a Seat and Seat should not extend Car here. Another general rule of Inheritance is that if you are creating a class which adds more feature into existing class, you can extend it to reuse all of its code. That’s the reason of using Runnable interface over Thread class for creating Thread in Java.

15.3 How to use Inheritance

You can use Inheritance in Java by using two keywords, extends and implements. extends keyword is used when one Class inherit from other Class or one interface extend another interface. On the other hand implements keyword is used when a class implement an interface which is also a form of Abstraction in Java. Interestingly, Inheritance facilitate Polymorphism in Java. By using Inheritance Sub class gets all property of Super class, except private, and can represent Super class i.e. you can store sub class instance in a Super class reference variable, which is a form of Polymorphism in Java. All flexibility which is provided by interface based design is achieved using polymorphism. Common example of this Factory design pattern in Java where return type of Factory method should be base interface, which allows Factory method to return any implementation of base interface, which is actually created using Inheritance in Java. In next section we will an example of Inheritance, which will show How to code for inheritance. in Java

15.4 Inheritance Example

Here is a simple example of Inheritance in Java where we have a class called Server which represent any Server has common functionality e.g. uptime(), start() and stop(). Since every Server has own way of starting and stopping, it can override those method but any Server class can reuse common code which is applicable to all type of Server e.g. uptime.

```
/**
 *
 * Java program to demonstrate Inheritance in Java programming
language.
 * Inheritance is used to reuse code and Java programming language
allows you
 * to either extend class or implements Interface. In this Program we
have a
 * Super Class Server and a Sub Class Tomcat, which is a Server.
 * Tomcat inherit start() and stop() method of Server Super Class.
 *
 * @author Javin
 */
public class Inheritance{

    public static void main(String args[]) {
        //Super class reference variable can hold Sub Class instance
        Server server = new Tomcat();

       //we need to cast to get actual Server instance back in
reference variable.
       Tomcat tomcat = (Tomcat) server;
       tomcat.start(); //starting Server

       System.out.println( "Uptime of Server in nano: " +
server.uptime());

       tomcat.stop();
    }
```

```
}

class Server{
    private long uptime;

    public void start(){
        uptime = System.nanoTime();
    }

    public void stop(){
        uptime = 0;
    }

    public long uptime(){
        return uptime;
    }
}

class Tomcat extends Server{

    @Override
    public void start(){
        super.start();
        //Tomcat Server specific task
        System.out.println("Tomcat Server started");
    }

    @Override
    public void stop(){
        super.stop(); //you can call super class method using super
keyword
        System.out.println("Tomcat Server Stopped");
    }
}

Output:
Tomcat Server started
Uptime of Server : 105898370823666
Tomcat Server Stopped
```

That's all on What is Inheritance in Java, When to use Inheritance and How to use Inheritance in Java programming language. If you are coming from C++ background than only surprise for you is that Java does not support multiple Inheritance, other than that Java Inheritance is similar to Inheritance in any other programming language e.g. C++. in Java

16. Overloading vs Overriding

Though the name of the method remains same in the case of both method overloading and overriding, main difference comes from the fact that method overloading is resolved during compile time, while method overriding is resolved at runtime. Also rules of overriding or overloading a method are different in Java. For example, a private, static and final method cannot be overriding in Java but you can still overload them. For overriding both name and signature of the method must remain same, but in for

overloading method, the signature must be different. Last but not the least difference between them is that call to overloaded methods are resolved using static binding while the call to overridden method is resolved using dynamic binding in Java.

By the way, Method overloading and method overriding in Java is two important concept in Java which allows Java programmer to declare method with same name but different behavior. Method overloading and method overriding is based on Polymorphism in Java.

In case of method overloading, method with same name co-exists in same class but they must have different method signature, while in case of method overriding, method with same name is declared in derived class or sub class. Method overloading is resolved using static binding in Java at compile time while method overriding is resolved using dynamic binding in Java at runtime.

In short, when you overload a method in Java its method signature got changed while in case of overriding method signature remains same but a method can only be overridden in sub class. Since Java supports Polymorphism and resolve object at run-time it is capable to call overridden method in Java.

16.1 Overloading and Overriding Key Points

1 private: A java private method cannot be overridden because, it is not accessible to an inheriting class.
2 final: Method overloading a Java final method is possible but overriding a final method is not allowed.
3 final: Two methods with same parameters list and in one method we have parameters 'final', in this case two methods are not different methods and overloading is not possible.
4 static: Method overloading a Java static method is possible but overriding a static method is not possible.
5 static: Overriding is in the context of objects (instances) and inheritance. A method declared as static belongs to the whole class (common for all objects). So there is no point of overriding it in sub-class.
6 static: In overloading if two methods are different only by a static modifier then its not possible to overload. The arguments list must be different.

16.2 Rules of Method Overriding

Following are rules of method overriding in java which must be followed while overriding any method. As stated earlier private, static and final method cannot be overridden in Java.

- Method signature must be same including return type, number of method parameters, type of parameters and order of parameters
- Overriding method cannot throw higher Exception than original or overridden method. means if original method throws IOException than overriding method cannot throw super class of IOException e.g. Exception but it can throw any sub class of IOException or simply does not throw any Exception. This rule only applies to checked Exception in Java, overridden method is free to throw any unchecked Exception.
- Overriding method cannot reduce accessibility of overridden method, means if original or overridden method is public than overriding method cannot make it protected.

16.3 Difference between Method Overloading vs overriding

1) First and most important difference between method overloading and overriding is that, In case of method overloading in Java, signature of method changes while in case of method overriding it remain same.

2) Second major difference between method overloading vs overriding in Java is that you can overload method in one class but overriding can only be done on subclass.

3) You cannot override static, final and private method in Java but you can overload static, final or private method in Java.

4) Overloaded method in Java is bonded by static binding and overridden methods are subject to dynamic binding.

5) Private and final method can also be not overridden in Java.

16.4 Handling Exception while overloading and overriding method

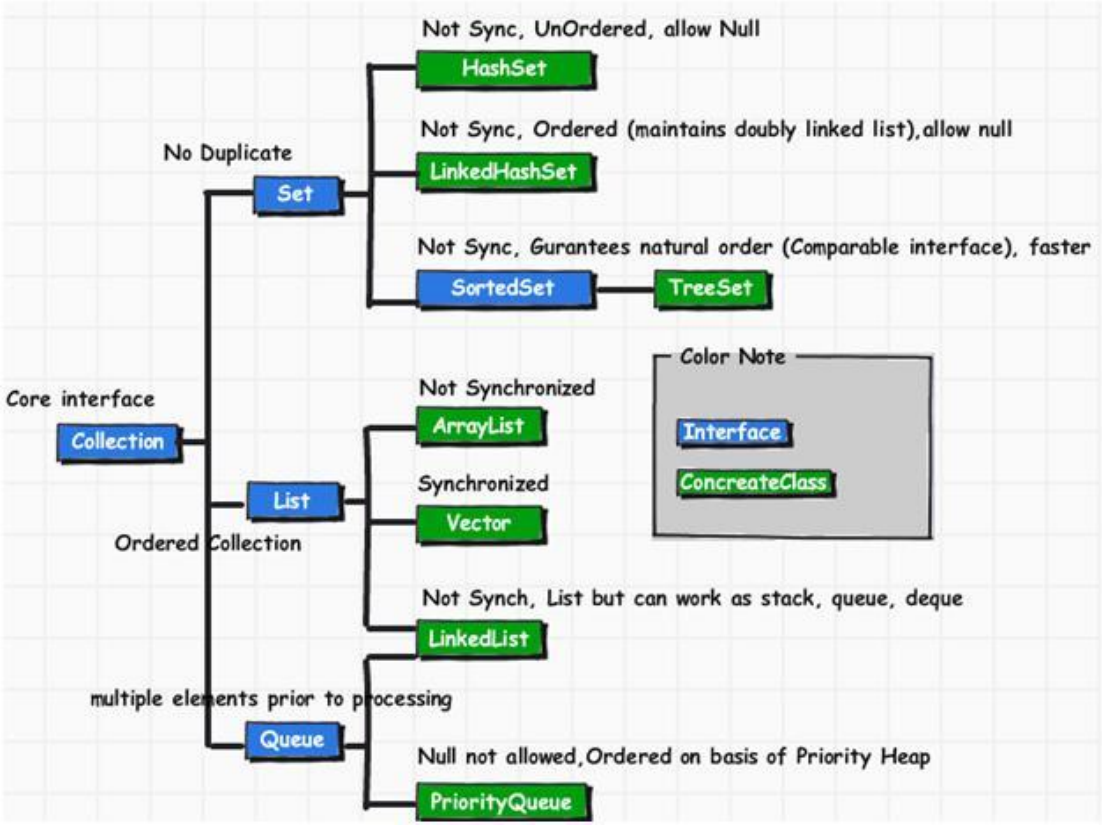
While overriding a method it can only throw checked exception declared by by overridden method or any subclass of it, means if overridden method throws IOExcpetion than overriding method can throw sub classes of IOExcpetion e.g. FileNotFoundException but not wider exception e.g. Exception or Throwable. This restriction is only for checked Exception for RuntimeException you can throw any RuntimeException. Overloaded method in Java doesn't have such restriction and you are free to modify throws clause as per your need.

16.5 Covariant Method Overriding

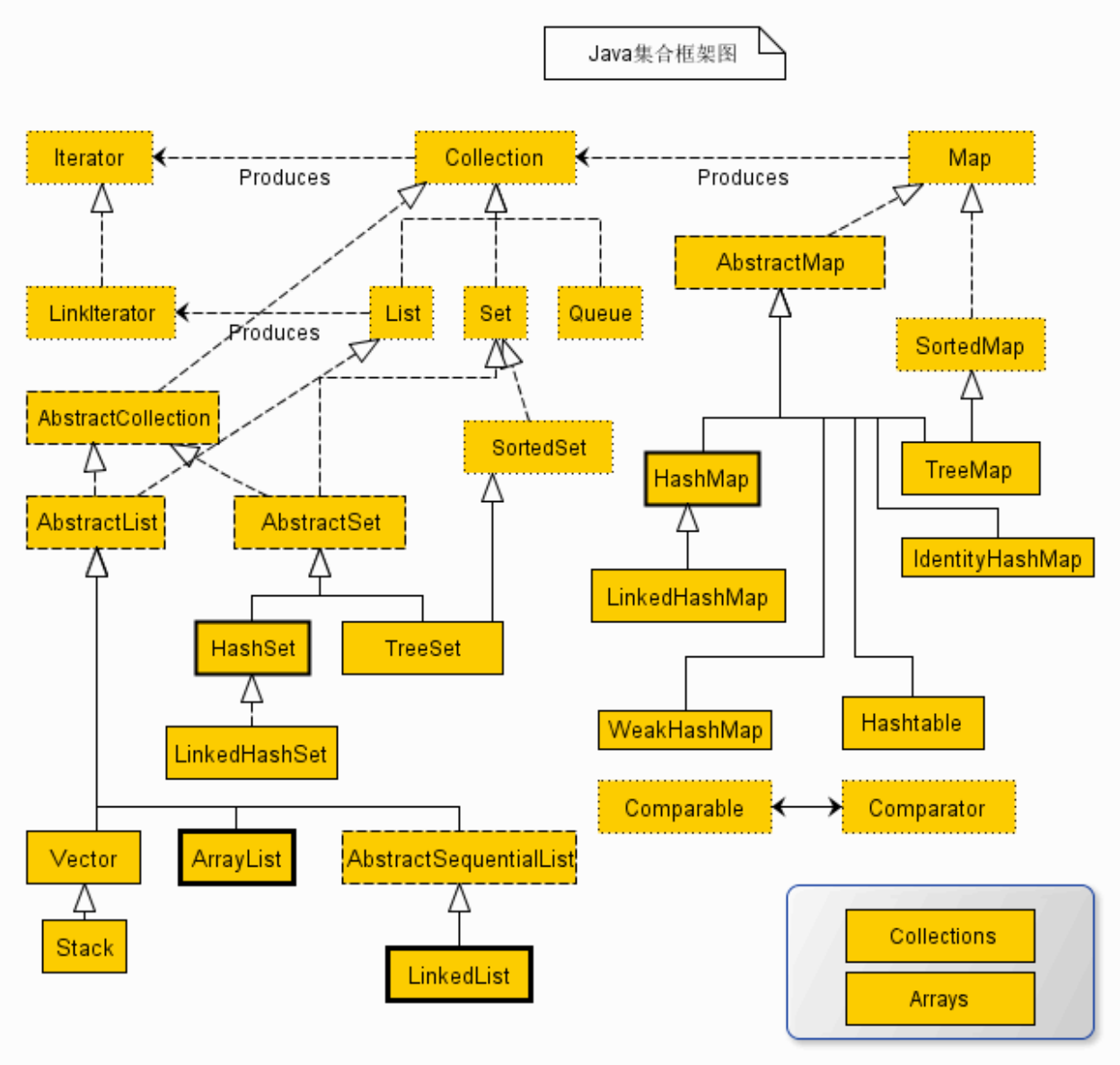
```
class Alpha {
    Alpha doStuff(char c) {
        return new Alpha();
    }
}

class Beta extends Alpha {
    Beta doStuff(char c) { // legal override in Java 1.5
        return new Beta();
    }
}
```

You can see that Beta class which is overriding doStuff() method from Alpha class is returning Beta type and not Alpha type. This will remove type casting on client side.

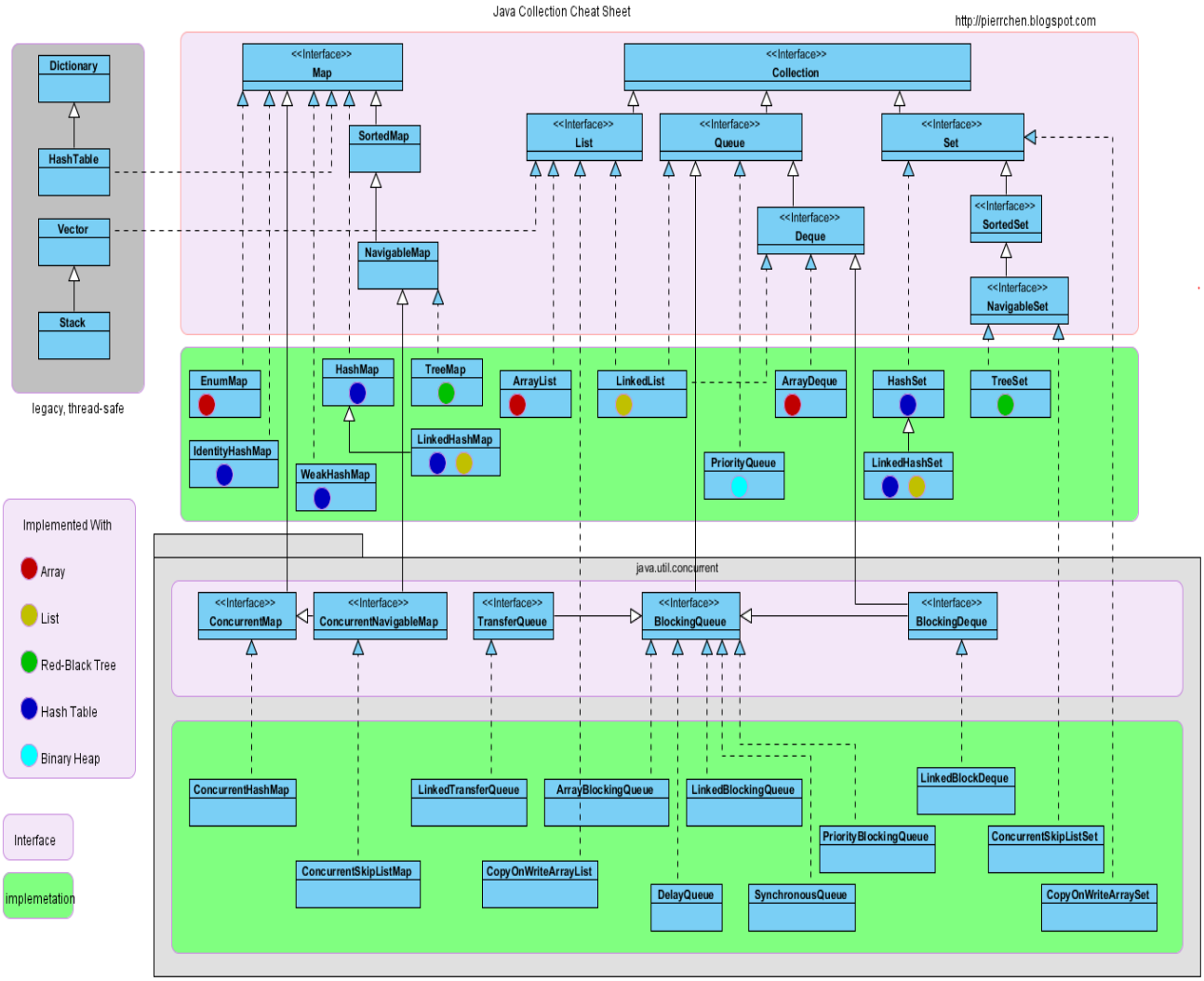
17. Java Collection



18. Java Collection Flow



19. Java Collection heat Sheet



20. Collection Big O Cheat Sheet

	get	add	contains	next	remove(O)	Iterator.remove
ArrayList	O(1)	O(1)	O(n)	O(1)	O(n)	O(n)
LinkedList	O(n)	O(1)	O(n)	O(1)	O(1)	O(1)
CopyOnWriteArrayList	O(1)	O(n)	O(n)	O(1)	O(n)	O(n)

	get	containsKey	next	Note
HashMap	O(1)	O(1)	O(h/n)	h is the table capacity
LinkedHashMap	O(1)	O(1)	O(1)	
IdentityHashMap	O(1)	O(1)	O(h/n)	h is the table capacity
EnumMap	O(1)	O(1)	O(1)	
TreeMap	O(log n)	O(log n)	O(log n)	
ConcurrentHashMap	O(1)	O(1)	O(h/n)	h is the table capacity
ConcurrentSkipListMap	O(log n)	O(log n)	O(1)	

	add	contains	next	Note
HashSet	O(1)	O(1)	O(h/n)	h is the table capacity
LinkedHashSet	O(1)	O(1)	O(1)	
CopyOnWriteArraySet	O(n)	O(n)	O(1)	
EnumSet	O(1)	O(1)	O(1)	
TreeSet	O(log n)	O(log n)	O(log n)	
ConcurrentSkipListSet	O(log n)	O(log n)	O(1)	

	offer	peek	poll	size
PriorityQueue	O(log n)	O(1)	O(log n)	O(1)
ConcurrentLinkedQueue	O(1)	O(1)	O(1)	O(n)
ArrayBlockingQueue	O(1)	O(1)	O(1)	O(1)
LinkedBlockingQueue	O(1)	O(1)	O(1)	O(1)
PriorityBlockingQueue	O(log n)	O(1)	O(log n)	O(1)
DelayQueue	O(log n)	O(1)	O(log n)	O(1)
LinkedList	O(1)	O(1)	O(1)	O(1)
ArrayDeque	O(1)	O(1)	O(1)	O(1)
LinkedBlockingDequeue	O(1)	O(1)	O(1)	O(1)

21. How HashMap internally works

Hash Map is one of the most used collection, though it will be surprising to know that maps themselves are not collections because they don't implement Collection interface. However collection view of a map can be obtained using entrySet() method. To obtain a collection-view of the keys, keySet() method can be used.

Coming to the internal working of the HashMap, which is also a favourite Java Collections interview question, there are four things we should know about before going into the internals of how does HashMap work in Java - in Java:

1 **HashMap** works on the principal of hashing.

2 **Map.Entry** interface - This interface gives a map entry (key-value pair). HashMap in Java stores both key and value object, in bucket, as an object of Entry class which implements this nested interface Map.Entry.

3 **hashCode()** -HashMap provides put(key, value) for storing and get(key) method for retrieving Values from HashMap. When put() method is used to store (Key, Value) pair, HashMap implementation calls hashcode on Key object to calculate a hash that is used to find a bucket where Entry object will be stored. When get() method is used to retrieve value, again key object is used to calculate a hash which is used then to find a bucket where that particular key is stored.

4 **equals()** - equals() method is used to compare objects for equality. In case of HashMap key object is used for comparison, also using equals() method Map knows how to handle hashing collision (hashing collision means more than one key having the same hash value, thus assigned to the same bucket. In that case objects are stored in a linked list, refer figure for more clarity.

Where hashCode method helps in finding the bucket where that key is stored, equals method helps in finding the right key as there may be more than one key-value pair stored in a single bucket.

** Bucket term used here is actually an index of array, that array is called table in HashMap implementation. Thus table[0] is referred as bucket0, table[1] as bucket1 and so on.

21.1 Understanding the Code

How important it is to have a proper hash code and equals method can be seen through the help of the following program –

```
public class HashMapTest {
    public static void main(String[] args) {
        Map <Key, String> cityMap = new HashMap<Key, String>();
        cityMap.put(new Key(1, "NY"),"New York City" );
        cityMap.put(new Key(2, "ND"), "New Delhi");
        cityMap.put(new Key(3, "NW"), "Newark");
        cityMap.put(new Key(4, "NP"), "Newport");

        System.out.println("size before iteration " + cityMap.size());
        Iterator <Key> itr = cityMap.keySet().iterator();
        while (itr.hasNext()){
            System.out.println(cityMap.get(itr.next()));
        }
        System.out.println("size after iteration " + cityMap.size());
    }

}

// This class' object is used as key
// in the HashMap
class Key{
 int index;
 String Name;
 Key(int index, String Name){
  this.index = index;
  this.Name = Name;
 }
```

```
 @Override
 // A very bad implementation of hashcode
 // done here for illustrative purpose only
 public int hashCode(){
  return 5;
 }

 @Override
 // A very bad implementation of equals
 // done here for illustrative purpose only
 public boolean equals(Object obj){
  return true;
 }

}
Output

size before iteration 1
Newport
size after iteration 1
```

lets get through the code to see what is happening, this will also help in understanding how put works internally.

Notice that I am inserting 4 values in the HashMap, still in the output it says size is 1 and iterating the map gives me the last inserted entry. Why is that?

Answer lies in, how hashCode() and equals() method are implemented for the key Class. Have a look at the hashCode() method of the class Key which always returns "5" and the equals() method which is always returning "true".

When a value is put into HashMap it calculates a hash using key object and for that it uses the hashCode() method of the key object class (or its parent class). Based on the calculated hash value HashMap implementation decides which bucket should store the particular Entry object.

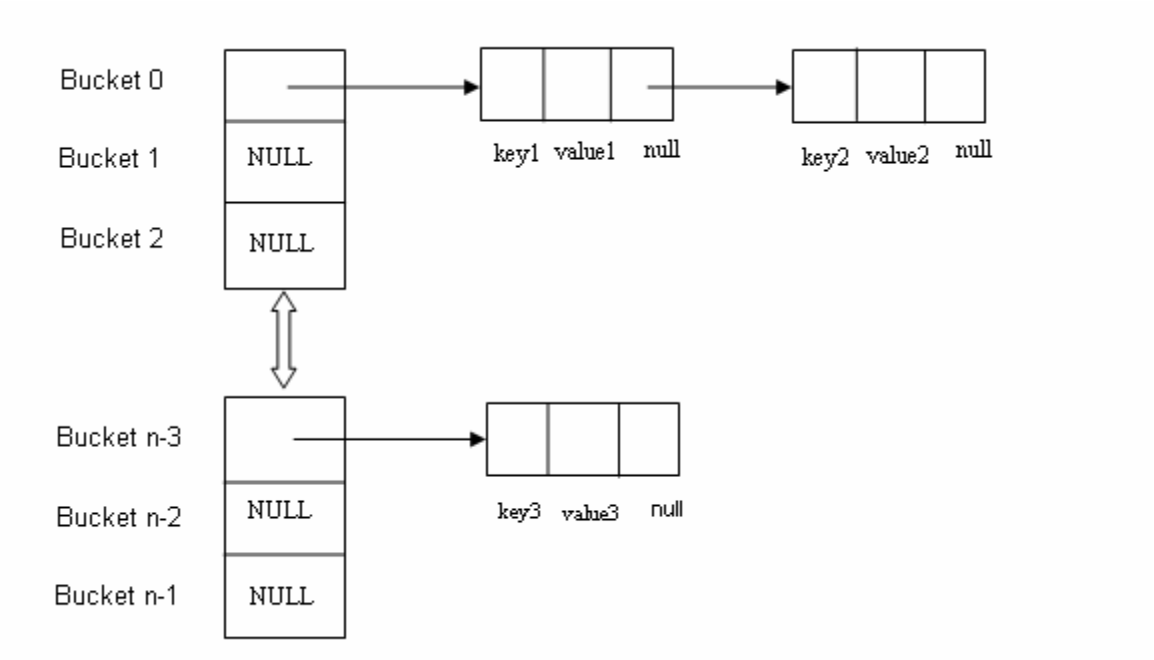
In my code the hashCode() method of the key class always returns "5". This effectively means calculated hash value, is same for all the entries inserted in the HashMap. Thus all the entries are stored in the same bucket.

Second thing, a HashMap implementation does is to use equals() method to see if the key is equal to any of the already inserted keys (Recall that there may be more than one entry in the same bucket). Note that, with in a bucket key-value pair entries (Entry objects) are stored in a linked-list (Refer figure for more clarity). In case hash is same, but equals() returns false (which essentially means more than one key having the same hash or hash collision) Entry objects are stored, with in the same bucket, in a linked-list.

In my code, I am always returning true for equals() method so the HashMap implementation "thinks" that the keys are equal and overwrites the value. So, in a way using hashCode() and equals() I have "tricked" HashMap implementation to think that all the keys (even though different) are same, thus overwriting the values.

In a nutshell there are three scenarios in case of put() –

- Using hashCode() method, hash value will be calculated. Using that hash it will be ascertained, in which bucket particular entry will be stored.
- equals() method is used to find if such a key already exists in that bucket, if no then a new node is created with the map entry and stored within the same bucket. A linked-list is used to store those nodes.
- If equals() method returns true, which means that the key already exists in the bucket. In that case, the new value will overwrite the old value for the matched key.



Pictorial representation of how Entry (key-value pair) objects will be stored in table array

21.2 How get() methods works internally

As we already know how Entry objects are stored in a bucket and what happens in the case of Hash Collision it is easy to understand what happens when key object is passed in the get method of the HashMap to retrieve a value.

Using the key again hash value will be calculated to determine the bucket where that Entry object is stored, in case there are more than one Entry object with in the same bucket stored as a linked-list equals() method will be used to find out the correct key. As soon as the matching key is found get() method will return the value object stored in the Entry object.

21.3 In case of null Key

As we know that HashMap also allows null, though there can only be one null key in HashMap. While storing the Entry object HashMap implementation checks if the key is null, in case key is null, it always map to bucket 0 as hash is not calculated for null keys.

21.4 HashMap changes in Java 8

Though HashMap implementation provides constant time performance O(1) for get() and put() method but that is in the ideal case when the Hash function distributes the objects evenly among the buckets.

But the performance may worsen in the case hashCode() used is not proper and there are lots of hash collisions. As we know now that in case of hash collision entry objects are stored as a node in a linked-list and equals() method is used to compare keys. That comparison to find the correct key with in a linked-list is a linear operation so in a worst case scenario the complexity becomes O(n).

To address this issue in Java 8 hash elements use balanced trees instead of linked lists after a certain threshold is reached. Which means HashMap starts with storing Entry objects in linked list but after the number of items in a hash becomes larger than a certain threshold, the hash will change from using a linked list to a balanced tree, this will improve the worst case performance from O(n) to O(log n).

Points to note -

- HashMap works on the principal of hashing.
- HashMap uses the hashCode() method to calculate a hash value. Hash value is calculated using the key object. This hash value is used to find the correct bucket where Entry object will be stored.
- HashMap uses the equals() method to find the correct key whose value is to be retrieved in case of get() and to find if that key already exists or not in case of put().
- Hashing collision means more than one key having the same hash value, in that case Entry objects are stored as a linked-list with in a same bucket.
- With in a bucket values are stored as Entry objects which contain both key and value.
- In Java 8 hash elements use balanced trees instead of linked lists after a certain threshold is reached while storing values. This improves the worst case performance from O(n) to O(log n).

22. Overriding hashCode() and equals() method

In mathematics, the Fibonacci numbers or Fibonacci sequence are the numbers in the following integer sequence:

22.1 Usage of hashCode() and equals()

equals() and hashCode() in Java are two methods which are present in the java.lang.Object class. These two methods are used for making inferences about an object's identity or in simpler language to reach to a decision whether the two compared objects are equal or not.

The default implementation of equals() method in the Object class is a simple reference equality check -

```
public boolean equals(Object obj){
 return (this == obj);
}
```

The default implementation of hashCode() in the Object class just returns integer value of the memory address of the object.

Usage of hashCode() and equals()

hashCode() - This method is used to get a unique integer value for a given object. We can see it's use with hash based collections like HashTable or HashMap where hashCode() is used to find the correct bucket location where the particular (key, value) pair is stored. Refer to How HashMap internally works

in Java to know more about it. equals() - equals() method is used to determine the equality of two objects.

22.2 When do we need to override hashCode() and equals()

It becomes very important to override these two methods in case we are using a custom object as key in a hash based collection. Also in case of ORM like Hibernate when a detached instance is re-attached to a session, we need to make sure that the object is same.

In such cases we can't rely on the default implementation provided by the Object class and need to provide custom implementation of hashCode() and equals() method.

But I never had the need to override these methods

Most of the time we use HashMap when it comes to hash based collections. Within HashMap as key, we mostly use primitive wrapper class like Integer or String class. These classes are immutable and provide their own proper implementation of hashCode() and equals(), thus these classes, on their own are, good hash keys.

At least I never had the need to implement these methods before I started using Hibernate. Things become tricky when we are using any custom object as key, in that case it becomes very important to override these 2 methods to make sure that we really get the object we want.

Let's say you are using some custom class object for a key in HashMap and that class did not override equals() and hashCode(), what will happen in that case?

In that case we would not be able to reliably retrieve the associated value using that key, unless we used the exact same class instance as key in the get() call as we did in the put() call. Why using the same instance it is possible? Because in the case we are not providing the implementation of equals() and hashCode(), Object class' equals() and hashCode() method will be used and recall that in Object class equals() method implementation is simple reference equality check thus having the same instance may satisfy the Object class' equals() method.

Let's see an example where custom object is used but hashCode and equals are not overridden and custom implementation is not given.

```
class Customer {
      private int customerID;
      private String firstName;
      private String lastName;

      public Customer(int customerID, String firstName, String
lastName) {
        super();
        this.customerID = customerID;
        this.firstName = firstName;
        this.lastName = lastName;
      }


}
public class HashCodeDemo {
```

```
    public static void main(String[] args) {
        Map<Customer, String> m = new HashMap<Customer, String>();
        Customer cust = new Customer(1, "Roger", "Cox");
        m.put(cust,"Roger Cox");
           // retrieving using another instance
        System.out.println(m.get(new Customer(1, "Roger", "Cox")));
           // retrieving using same instance
        System.out.println(m.get(cust));
    }
}
Output

null
Roger Cox
```

It can be seen how; when the instance is changed the value is not retrieved even when the same values for the customer object are given as key. But when we use the same instance then it is able to fetch. But that's not very convenient and in a big application we are not expected to keep the same instance and use it whenever we want to retrieve the value. So enter the equals() and hashCode() methods.

Same example with hashCode and equals implemented

```
class Customer {
      private int customerID;
      private String firstName;
      private String lastName;

      public Customer(int customerID, String firstName, String
lastName) {
        super();
        this.customerID = customerID;
        this.firstName = firstName;
        this.lastName = lastName;
      }

      @Override
        public int hashCode() {
            final int prime = 31;
            int result = 1;
            result = prime * result + customerID;
            result = prime * result
                    + ((firstName == null) ? 0 :
firstName.hashCode());
            result = prime * result
                    + ((lastName == null) ? 0 : lastName.hashCode());
            return result;
        }

        @Override
        public boolean equals(Object obj) {
            if (this == obj)
                return true;
            if (obj == null)
```

```
                return false;
            if (getClass() != obj.getClass())
                return false;
            Customer other = (Customer) obj;
            if (customerID != other.customerID)
                return false;
            if (firstName == null) {
                if (other.firstName != null)
                    return false;
            } else if (!firstName.equals(other.firstName))
                return false;
            if (lastName == null) {
                if (other.lastName != null)
                    return false;
            } else if (!lastName.equals(other.lastName))
                return false;
            return true;
        }


}
public class HashCodeDemo {

    public static void main(String[] args) {
        Map<Customer, String> m = new HashMap<Customer, String>();
        Customer cust = new Customer(1, "Roger", "Cox");
        m.put(cust,"Roger Cox");
        // retrieving using another instance
        System.out.println(m.get(new Customer(1, "Roger", "Cox")));
        // retrieving using same instance
        System.out.println(m.get(cust));
    }
}
Output

Roger Cox
Roger Cox
```

22.3 Requirements for implementing equals() and hashCode()

There are certain restrictions on the behavior of equals() and hashCode() methods, which can be seen in the Javadocs for Object class

For equals() method the JavaDocs say –

1. It is reflexive: for any non-null reference value x, x.equals(x) should return true.
2. It is symmetric: for any non-null reference values x and y, x.equals(y) should return true if and only if y.equals(x) returns true.
3. It is transitive: for any non-null reference values x, y, and z, if x.equals(y) returns true and y.equals(z) returns true, then x.equals(z) should return true.
4. It is consistent: for any non-null reference values x and y, multiple invocations of x.equals(y) consistently return true or consistently return false, provided no information used in equals comparisons on the objects is modified.

5. For any non-null reference value x, x.equals(null) should return false.

And the general contract of hashCode is –

1. Whenever it is invoked on the same object more than once during an execution of a Java application, the hashCode method must consistently return the same integer, provided no information used in equals comparisons on the object is modified. This integer need not remain consistent from one execution of an application to another execution of the same application.
2. If two objects are equal according to the equals(Object) method, then calling the hashCode method on each of the two objects must produce the same integer result.
3. It is not required that if two objects are unequal according to the equals(java.lang.Object) method, then calling the hashCode method on each of the two objects must produce distinct integer results. However, the programmer should be aware that producing distinct integer results for unequal objects may improve the performance of hash tables.

To explain this contract a little more -

The first point says that the hashCode() method must consistently return the same integer. In case a mutable object is used as key we have to make sure that its state does not change. If a key's hashCode changes while used in a Collection get and put will give some unpredictable results.

According to the second point if 2 objects Obj1 and Obj2 are equal according to their equals() method then they must have the same hash code too. Though the vice-versa is not true that is if 2 objects have the same hash code then they do not have to be equal too.

According to the third point if 2 objects are not equal according to their equals() method they still can have the same hash code. That is known as hash collision.

23. How HashSet works internally

Before going into internal working of Hashset it is important to know two points about HashSet.

- HashSet only stores unique values i.e. no duplicates are allowed.
- HashSet works on the concept of hashing too just like HashMap but how it differs from the HashMap is; in HashMap a Key, Value pair is added and the hash function is calculated using key. Where as in the HashSet hash function is calculated using the value itself. Note that in HashSet we have add(E e) method which takes just the element to be added as parameter.

Also you may have guessed by now, since hash function is calculated using value that is why only unique values are stored in the HashSet. If you try to store the same element again, the calculated hash function would be same, thus the element will be overwritten.

Now coming back to internal working of HashSet, which is also an important Java Collections interview question, the most important point is HashSet internally uses HashMap to store it's objects. Within the HashSet there are many constructors one without any parameter and several more with initial capacity or load factor but every one of these constructor creates a HashMap. If you know how HashMap works internally in Java it is easy to understand how HashSet works internally.

HashSet Constructor examples

```
/**
* Constructs a new, empty set; the backing <tt>HashMap</tt> instance
has
* default initial capacity (16) and load factor (0.75).
*/
public HashSet() {
    map = new HashMap<>();
}
public HashSet(int initialCapacity, float loadFactor) {
    map = new HashMap<>(initialCapacity, loadFactor);
}
```

And the map, which is used for storing values, is defined as

```
private transient HashMap<E,Object> map;
```

In the constructor, if you have noticed, there are parameters named initial capacity and load factor. For HashSet, default initial capacity is 16, that is an array (or bucket) of length 16 would be created and default load factor is 0.75. Where load factor is a measure of how full the hash table is allowed to get before its capacity is automatically increased.

23.1 How elements are added in HashSet

I have stated in the point 2 above that HashSet calculates the hash function using value itself and there is no (Key, Value) pair in HashSet and then comes the point that HashSet internally uses HashMap to store objects. These two statements may sound contradictory but if you know How HashMap internally works in Java it will be easy for you to understand how both of these these two contradictory statements hold true.

Actually in HashSet, from add method, put method of HashMap is called where the value which has to be added in the Set becomes Key and a constant object "PRESENT" is used as value.

That's how PRESENT is defined in HashSet implementation -

// Dummy value to associate with an Object in the backing Map

```
private static final Object PRESENT = new Object();
```

And that's how add method is implemented in HashSet class -

```
public boolean add(E e) {
    return map.put(e, PRESENT)==null;
}
```

One thing to note here is, in HashMap value may be duplicate but Key should be unique. That's how HashSet makes sure that only unique values are stored in it, since the value which is to be stored in the HashSet becomes the key while storing it in HashMap.

23.2 How Object is removed from a HashSet

When we need to remove an element from the HashSet, internally again remove method of HashSet calls remove(Object key) method of the HashMap.

That is how it is implemented in HashSet class.

```
public boolean remove(Object o) {
    return map.remove(o)==PRESENT;
}
```

Here note that remove(Object key) method of the HashMap returns the Value associated with the key. Whereas the remove(Object o) method of the HashSet returns Boolean value. Also we know that for every value added in HashSet that value becomes Key and the value is always an object called PRESENT. Therefore the value that is returned from the remove(Object key) method of the HashMap is always PRESENT thus the condition map.remove(o)==PRESENT.

23.3 How Object is retrieved from HashSet

In HashSet there is no get method as provided in Map or List. In HashSet iterator is there which will iterate through the values of the Set. Internally it will call the keyset of the HashMap, as values are stored as keys in the HashMap so what we'll get is the values stored in the HashSet.

That's how iterator is implemented in the HashSet

```
/**
* Returns an iterator over the elements in this set.  The elements
* are returned in no particular order.
*
* @return an Iterator over the elements in this set
* @see ConcurrentModificationException
*/
public Iterator<E> iterator() {
    return map.keySet().iterator();
}
```

24. Iterate over HashMap, Hashtable or any Map

There are multiple way to iterate, traverse or loop through Map, HashMap or TreeMap in Java. We will use following hashmap for our example:

```
HashMap<String, String> loans = new HashMap<String, String>();
loans.put<"home loan", "citibank");
loans.put<"personal loan", "Wells Fargo");
```

24.1 Iterating or looping map using Java5 foreach loop

Here we will use new foreach loop introduced in JDK5 for iterating over any map in java and using KeySet of map for getting keys. this will iterate through all values of Map and display key and value together.

```
HashMap<String, String> loans = new HashMap<String, String>();
loans.put("home loan", "citibank");
loans.put("personal loan", "Wells Fargo");

for (String key : loans.keySet()) {
```

```
   System.out.println("-----------------------------------------------
-");
   System.out.println("Iterating or looping map using java5 foreach
loop");
   System.out.println("key: " + key + " value: " + loans.get(key));
}

Output:
-----------------------------------------------
Iterating or looping map using java5 foreach looop
key: home loan value: citibank
-----------------------------------------------
Iterating or looping map using java5 foreach looop
key: personal loan value: Wells Fargo
```

24.2 Iterating Map in Java using KeySet Iterator

In this Example of looping hashmap in Java we have used Java Iterator instead of for loop, rest are similar to earlier example of looping:

```
Set<String> keySet = loans.keySet();
Iterator<String> keySetIterator = keySet.iterator();
while (keySetIterator.hasNext()) {
   System.out.println("-----------------------------------------------
-");
   System.out.println("Iterating Map in Java using KeySet Iterator");
   String key = keySetIterator.next();
   System.out.println("key: " + key + " value: " + loans.get(key));
}

Output:
-----------------------------------------------
Iterating Map in Java using KeySet Iterator
key: home loan value: citibank
-----------------------------------------------
Iterating Map in Java using KeySet Iterator
key: personal loan value: Wells Fargo
```

24.3 Looping HashMap in Java using EntrySet and Java 5 for loop

In this Example of traversing Map in Java, we have used EntrySet instead of KeySet. EntrySet is a collection of all Map Entries and contains both Key and Value.

```
Set<Map.Entry<String, String>> entrySet = loans.entrySet();
for (Entry entry : entrySet) {
   System.out.println("-----------------------------------------------
-");
   System.out.println("looping HashMap in Java using EntrySet and
java5 for loop");
   System.out.println("key: " + entry.getKey() + " value: " +
entry.getValue());
}
```

```
Output:
-----------------------------------------------
looping HashMap in Java using EntrySet and java5 for loop
key: home loan value: citibank
-----------------------------------------------
looping HashMap in Java using EntrySet and java5 for loop
key: personal loan value: Wells Fargo
```

24.4 Iterating HashMap in Java using EntrySet and Java iterator

This is the fourth and last example of looping Map and here we have used Combination of Iterator and EntrySet to display all keys and values of a Java Map.

```
Set<Map.Entry<String, String>> entrySet1 = loans.entrySet();
Iterator<Entry<String, String>> entrySetIterator =
entrySet1.iterator();
while (entrySetIterator.hasNext()) {
   System.out.println("-----------------------------------------");
   System.out.println("Iterating HashMap in Java using EntrySet and
Java iterator");
   Entry entry = entrySetIterator.next();
   System.out.println("key: " + entry.getKey() + " value: " +
entry.getValue());
}

Output:
-----------------------------------------------
Iterating HashMap in Java using EntrySet and Java iterator
key: home loan value: citibank
-----------------------------------------------
Iterating HashMap in Java using EntrySet and Java iterator
key: personal loan value: Wells Fargo
```

24.5 Java 8 – lambda method

```
public class CrunchifyJava8ForEachTutorialMapAndList {
	public static void main(String[] args) {
		// =============== MAP ================
		Map<String, String> crunchifyCompanyMap = new
HashMap<>();
		crunchifyCompanyMap.put("Google", "Mountain View");
		crunchifyCompanyMap.put("Facebook", "Santa Clara");
		crunchifyCompanyMap.put("Twitter", "San Francisco");

		// Method1: Standard Method to iterate through Java Map
		CrunchifyStandardForEachMethod4Map(crunchifyCompanyMap);
		// Method2: Java8 Method to iterate through Java Map
		CrunchifyJava8ForEachMethod4Map(crunchifyCompanyMap);

		// =============== List ================
		List<String> crunchifyCompanyList = new ArrayList<>();
		crunchifyCompanyList.add("Google");
```

```
                crunchifyCompanyList.add("Facebook");
                crunchifyCompanyList.add("Twitter");

                // Method3: Standard Method to iterate through Java List

        CrunchifyStandardForEachMethod4List(crunchifyCompanyList);
                // Method4,5: Java8 Method to iterate through Java List
                CrunchifyJava8ForEachMethod4List(crunchifyCompanyList);
        }
        private static void CrunchifyStandardForEachMethod4Map(
                        Map<String, String> crunchifyCompanyMap) {
                for (Map.Entry<String, String> crunchifyEntry :
crunchifyCompanyMap.entrySet()) {
                        log("crunchifyCompany: " + crunchifyEntry.getKey()
+ ", address: "
                                        + crunchifyEntry.getValue());
                }
        }
        private static void CrunchifyJava8ForEachMethod4Map(Map<String,
String> crunchifyCompanyMap) {
                crunchifyCompanyMap.forEach((k, v) ->
log("crunchifyCompany: " + k + ", address: " + v));
        }
        private static void
CrunchifyStandardForEachMethod4List(List<String> crunchifyList) {
                for (String item : crunchifyList) {
                        log(item);
                }
        }

        private static void
CrunchifyJava8ForEachMethod4List(List<String> crunchifyList) {
                // lambda method
                crunchifyList.forEach(item -> log(item));

                // using method reference
                crunchifyList.forEach(System.out::println);

        }
        // Simple log utility
        private static void log(String string) {
                System.out.println(string);
        }
}
```

25. ConcurrentHashMap, Hashtable and Synchronized Map

ConcurrentHashMap allows concurrent modification of the Map from several threads without the need to block them. Collections.synchronizedMap(map) creates a blocking Map which will degrade performance, albeit ensure consistency (if used properly).

Use the second option if you need to ensure data consistency, and each thread needs to have an up-to-date view of the map. Use the first if performance is critical, and each thread only inserts data to the map, with reads happening less frequently.

Though all three collection classes are thread-safe and can be used in multi-threaded, concurrent Java application, there is a significant difference between them, which arise from the fact that how they achieve their thread-safety. Hashtable is a legacy class from JDK 1.1 itself, which uses synchronized methods to achieve thread-safety. All methods of Hashtable are synchronized which makes them quite slow due to contention if a number of thread increases. Synchronized Map is also not very different than Hashtable and provides similar performance in concurrent Java programs. The only difference between Hashtable and Synchronized Map is that later is not a legacy and you can wrap any Map to create it's synchronized version by using Collections.synchronizedMap() method.

On the other hand, ConcurrentHashMap is specially designed for concurrent use i.e. more than one thread. By default it simultaneously allows 16 threads to read and write from Map without any external synchronization. It is also very scalable because of stripped locking technique used in the internal implementation of ConcurrentHashMap class. Unlike Hashtable and Synchronized Map, it never locks whole Map, instead, it divides the map into segments and locking is done on those. Though it performs better if a number of reader threads are greater than the number of writer threads.

To be frank, Collections classes are the heart of Java API though I feel using them judiciously is an art. It's my personal experience where I have improved the performance of Java application by using ArrayList where legacy codes were unnecessarily using Vector etc. Prior Java 5, One of the major drawback of Java Collection framework was a lack of scalability.

In multi-threaded Java application synchronized collection classes like Hashtable and Vector quickly becomes the bottleneck; to address scalability JDK 1.5 introduces some good concurrent collections which are highly efficient for high volume, low latency system electronic trading systems In general those are the backbone for Concurrent fast access to stored data.

In this tutorial, we will look on ConcurrentHashMap, Hashtable, HashMap and synchronized Map and see the difference between ConcurrentHashMap and Hashtable and synchronized Map in Java. We have already discussed some key difference between HashMap and Hashtable in Java in this blog and those will also help you to answer this question during interviews.

25.1 Why need ConcurrentHashMap and CopyOnWriteArrayList

The synchronized collections classes, Hashtable, and Vector, and the synchronized wrapper classes, Collections.synchronizedMap() and Collections.synchronizedList(), provide a basic conditionally thread-safe implementation of Map and List. However, several factors make them unsuitable for use in highly concurrent applications, for example, their single collection-wide lock is an impediment to scalability and it often becomes necessary to lock a collection for a considerable time during iteration to prevent ConcurrentModificationException.

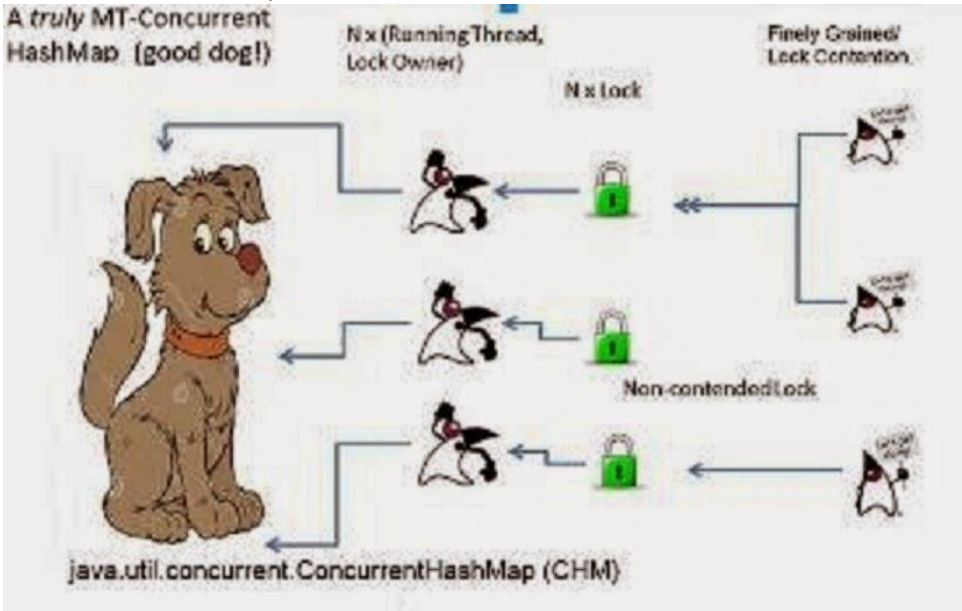
ConcurrentHashMap and CopyOnWriteArrayList implementations provide much higher concurrency while preserving thread safety, with some minor compromises in their promises to callers. ConcurrentHashMap and CopyOnWriteArrayList are not necessarily useful everywhere you might use HashMap or ArrayList, but are designed to optimize specific common situations. Many concurrent applications will benefit from their use.

25.2 Difference between ConcurrentHashMap and Hashtable

So what is the difference between Hashtable and ConcurrentHashMap, both can be used in the multithreaded environment but once the size of Hashtable becomes considerable large performance degrade because for iteration it has to be locked for a longer duration.

Since ConcurrentHashMap introduced the concept of segmentation, how large it becomes only certain part of it get locked to provide thread safety so many other readers can still access map without waiting for iteration to complete.

In Summary, ConcurrentHashMap only locked certain portion of Map while Hashtable locks full map while doing iteration. This will be clearer by looking at this diagram which explains the internal working of ConcurrentHashMap in Java.



25.3 The difference between ConcurrentHashMap and Collections.synchronizedMap

ConcurrentHashMap is designed for concurrency and improve performance while HashMap which is non-synchronized by nature can be synchronized by applying a wrapper using synchronized Map. Here are some of the common differences between ConcurrentHashMap and synchronized map in Java

ConcurrentHashMap does not allow null keys or null values while synchronized HashMap allows one null key.

25.4 Difference between ConcurrentHashMap and HashMap

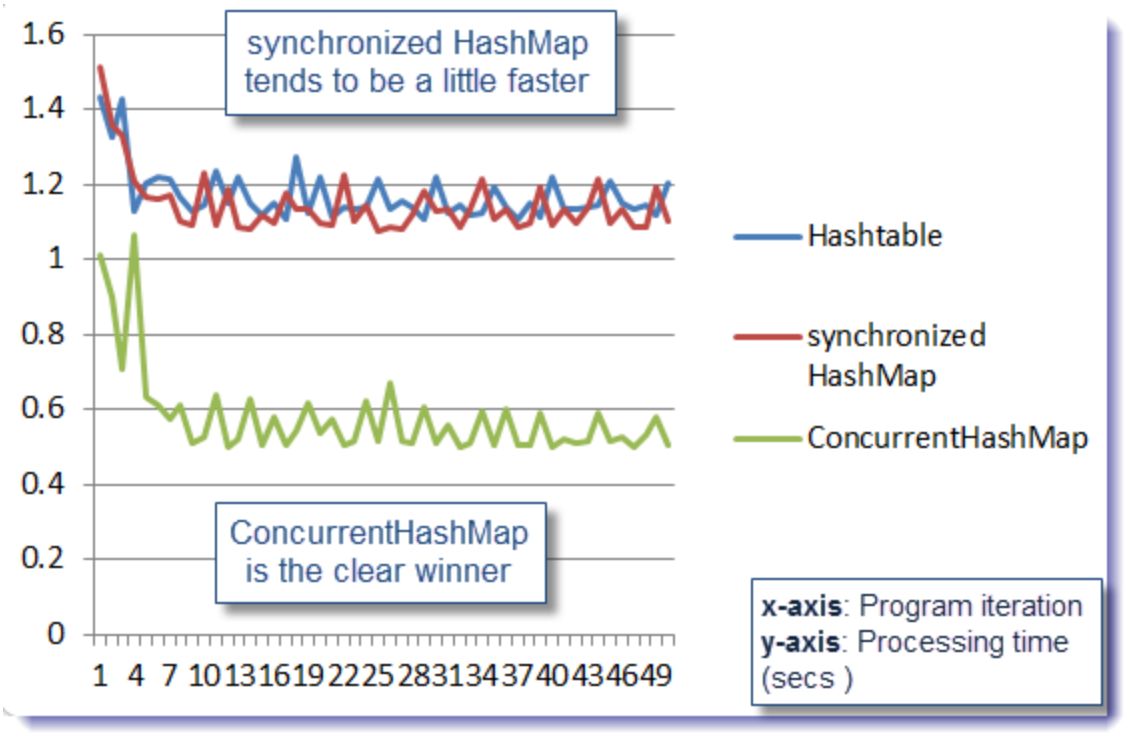
1. First and foremost difference is of course thread safety. ConcurrentHashMap is thread safe and fit for use in a multi-threaded environment whereas HashMap is not thread safe.
2. Second difference is about how these data structures synchronize. HashMap can be synchronized using the Collections.synchronizedMap() method but that synchronizes all the methods of the HashMap and effectively reduces it to a data structure where one thread can enter at a time.

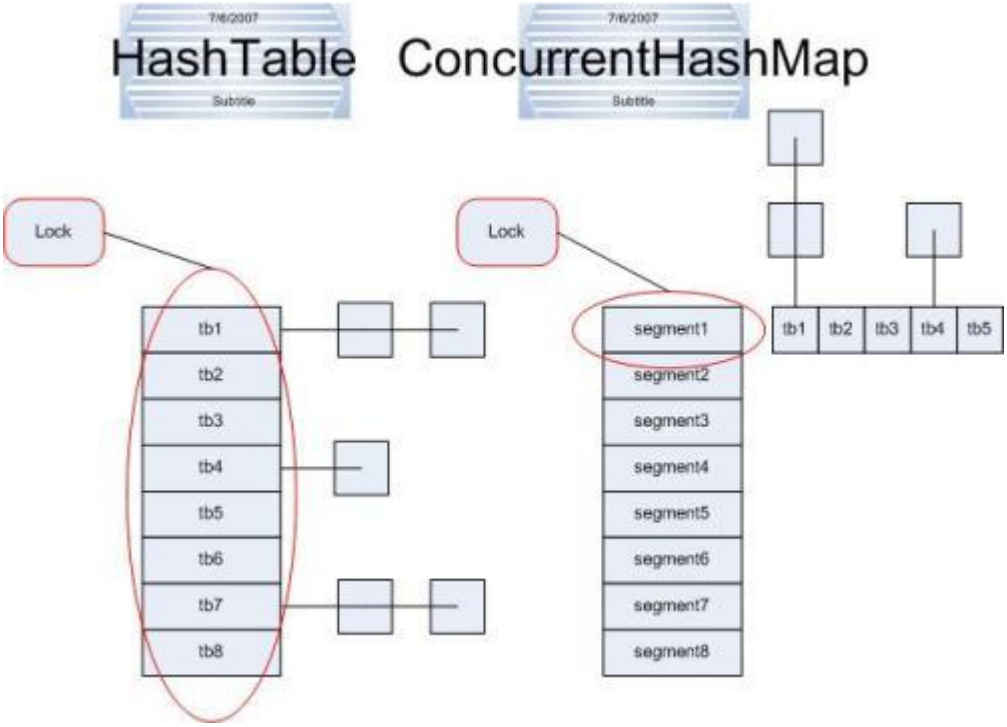
3. In ConcurrentHashMap synchronization is done a little differently. Rather than locking every method on a common lock, ConcurrentHashMap uses separate lock for separate buckets thus locking only a portion of the Map.
4. By default there are 16 buckets and also separate locks for separate buckets. So the default concurrency level is 16. That means theoretically any given time 16 threads can access ConcurrentHashMap if they all are going to separate buckets.
5. In ConcurrentHashMap performance is further improved by providing read access concurrently without any blocking. Retrieval operations (including get) generally do not block, so may overlap with update operations (including put and remove).
6. HashMap allows one null as key but ConcurrentHashMap doesn't allow null as key.
7. Performace wise HashMap is better as there is no synchronization.
8. In case HashMap has to be used in a multi-threaded environment and there is a need to use Collections.SynchronizedMap() method then ConcurrentHashMap() is a better choice as ConcurrentHashMap still gives a chance to more than one thread to access map thus improving performance.
9. Iterator provided by ConcurrentHashMap is fail-safe which means it will not throw ConcurrentModificationException if the underlying structure is changed during iteration.
10. Iterator provided by HashMap is fail-fast as it throws a ConcurrentModificationException if the underlying collection is structurally modified at any time after the iterator is created.

26. ConcurrentHashMap in Java

Though we have an option to synchronize the collections like List or Set using synchronizedList or synchronizedSet method respectively of the Collections class but there is a drawback of this synchronization; very poor performance as the whole collection is locked and only a single thread can access it at a given time. On the other hand there is HashTable too which is thread safe but that thread safety is achieved by having all its method as synchronized which again results in poor performance.

From Java 5 ConcurrentHashMap is introduced as another alternative for HashTable(or explicitly synchronizing the map using synchronizedMap method).





26.1 Why another Map

First thing that comes to mind is why another map when we already have HashMap or HashTable (if thread safe data structure is required). The answer is better performance though still providing a thread safe alternative. So it can be said that ConcurrentHashMap is a hash map whose operations are threadsafe.

26.2 How performance is improved

As I said ConcurrentHashMap is also a hash based map like HashMap, how it differs is the locking strategy used by ConcurrentHashMap. Unlike HashTable (or synchronized HashMap) it doesn't synchronize every method on a common lock. ConcurrentHashMap uses separate lock for separate buckets thus locking only a portion of the Map. Just for information ConcurrentHashMap uses ReentrantLock for locking

If you have idea about the internal implementation of the HashMap you must be knowing that by default there are 16 buckets. Same concept is used in ConcurrentHashMap and by default there are 16 buckets and also separate locks for separate buckets. So the default concurrency level is 16.

Which is effectively this constructor

ConcurrentHashMap() Creates a new, empty map with a default initial capacity (16), load factor (0.75) and concurrencyLevel (16).There are several other constructors, one of them is

```
ConcurrentHashMap(int initialCapacity, float loadFactor, int
concurrencyLevel)
```

Creates a new, empty map with the specified initial capacity, load factor and concurrency level. So you can see that you can provide your own initial capacity (default is 16), own load factor and also concurrency level (Which by default is 16).

Since there are 16 buckets having separate locks of their own which effectively means at any given time 16 threads can operate on the map concurrently provided all these threads are operating on separate buckets.

So you see how ConcurrentHashMap provides better performance by locking only the portion of the map rather than blocking the whole map resulting in greater shared access.

That is not all, performance is further improved by providing read access concurrently without any blocking. Retrieval operations (including get) generally do not block, so may overlap with update operations (including put and remove). Retrievals reflect the results of the most recently completed update operations which may mean that retrieval operations may not fetch the current/in-progress value (Which is one drawback). Memory visibility for the read operations is ensured by volatile reads. You can see in the ConcurrentHashMap code that the val and next fields of Node are volatile.

Also for aggregate operations such as putAll and clear which works on the whole map, concurrent retrievals may reflect insertion or removal of only some entries (another drawback of separate locking). Because read operations are not blocking but some of the writes (which are on the same bucket) may still be blocking.

26.3 Simple example using ConcurrentHashMap

At this juncture let's see a simple example of ConcurrentHashMap.

```
public class CHMDemo {

    public static void main(String[] args) {
        // Creating ConcurrentHashMap
        Map<String, String> cityTemperatureMap = new
ConcurrentHashMap<String, String>();
```

```
        // Storing elements
        cityTemperatureMap.put("Delhi", "24");
        cityTemperatureMap.put("Mumbai", "32");
        //cityTemperatureMap.put(null, "26");
        cityTemperatureMap.put("Chennai", "35");
        cityTemperatureMap.put("Bangalore", "22" );

        for (Map.Entry e : cityTemperatureMap.entrySet()) {
            System.out.println(e.getKey() + " = " + e.getValue());
        }
    }
}
```

26.4 Null is not allowed

Though HashMap allows one null as key but ConcurrentHashMap doesn't allow null as key. In the previous example you can uncomment the line which has null key. While trying to execute the program it will throw null pointer exception.

```
Exception in thread "main" java.lang.NullPointerException
 at java.util.concurrent.ConcurrentHashMap.putVal(Unknown Source)
 at java.util.concurrent.ConcurrentHashMap.put(Unknown Source)
 at org.netjs.prog.CHMDemo.main(CHMDemo.java:16)
```

26.5 Atomic operations

ConcurrentHashMap provides a lot of atomic methods, let's see it with an example how these atomic methods help. Note that from Java 8 many new atomic methods are added.

Suppose you have a word Map that counts the frequency of every word where key is the word and count is the value, in a multi-threaded environment, even if ConcurrentHashMap is used, there may be a problem as described in the code snippet.

```
public class CHMAtomicDemo {

    public static void main(String[] args) {
        ConcurrentHashMap<String, Integer> wordMap = new
ConcurrentHashMap<>();
        ..
        ..
        // Suppose one thread is interrupted after this line and
        // another thread starts execution
        Integer prevValue = wordMap.get(word);

        Integer newValue = (prevValue == null ? 1 : prevValue + 1);
        // Here the value may not be correct after the execution of
        // both threads
        wordMap.put(word, newValue);

    }
```

```
}
```

To avoid these kind of problems you can use atomic method, one of the atomic method is compute which can be used here.

```
wordMap.compute(word, (k,v)-> v == null ? 1 : v + 1);
```

If you see the general structure of the Compute method

```
compute(K key, BiFunction<? super K,? super V,? extendsV>
```

remappingFunction) Here BiFunction functional interface is used which can be implemented as a lambda expression.

So here rather than having these lines -

```
Integer prevValue = wordMap.get(word);
Integer newValue = (prevValue == null ? 1 : prevValue + 1);
wordMap.put(word, newValue);
```

you can have only this line

```
wordMap.compute(word, (k,v)-> v == null ? 1 : v + 1);
```

The entire method invocation is performed atomically. Some attempted update operations on this map by other threads may be blocked while computation is in progress.

There are several other atomic operations like computeIfAbsent, computeIfPresent, merge, putIfAbsent.

26.6 Fail-safe iterator

The iterator generated by the ConcurrentHashMap is fail-safe which means it will not throw ConcurrentModificationException.

Example code

```
public class CHMDemo {

    public static void main(String[] args) {
        // Creating ConcurrentHashMap
        Map<String, String> cityTemperatureMap = new
ConcurrentHashMap<String, String>();

        // Storing elements
        cityTemperatureMap.put("Delhi", "24");
        cityTemperatureMap.put("Mumbai", "32");
        cityTemperatureMap.put("Chennai", "35");
        cityTemperatureMap.put("Bangalore", "22" );

        Iterator<String> iterator =
cityTemperatureMap.keySet().iterator();
        while (iterator.hasNext()){

System.out.println(cityTemperatureMap.get(iterator.next()));
            // adding new value, it won't throw error
            cityTemperatureMap.put("Kolkata", "34");

        }
    }
```

```
}
Output

24
35
34
32
22
```

According to the JavaDocs - The view's iterator is a "weakly consistent" iterator that will never throw ConcurrentModificationException, and guarantees to traverse elements as they existed upon construction of the iterator, and may (but is not guaranteed to) reflect any modifications subsequent to construction.

26.7 When ConcurrentHashMap is a better choice

ConcurrentHashMap is a better choice when there are more reads than writes. As mentioned above retrieval operations are non-blocking so many concurrent threads can read without any performance problem. If there are more writes and that too many threads operating on the same segment then the threads will block which will deteriorate the performance.

26.8 Points to note

- ConcurrentHashMap is also a hash based map like HashMap, but ConcurrentHashMap is thread safe.
- In ConcurrentHashMap thread safety is ensured by having separate locks for separate buckets, resulting in better performance.
- By default the bucket size is 16 and the concurrency level is also 16.
- No null keys are allowed in ConcurrentHashMap.
- Iterator provided by the ConcurrentHashMap is fail-safe, which means it will not throw ConcurrentModificationException.
- Retrieval operations (like get) don't block so may overlap with update operations (including put and remove).

27. Java Class Loading

Java class loaders are used to load classes at runtime. ClassLoader in Java works on three principle: delegation, visibility and uniqueness. Delegation principle forward request of class loading to parent class loader and only loads the class, if parent is not able to find or load class. Visibility principle allows child class loader to see all the classes loaded by parent ClassLoader, but parent class loader can not see classes loaded by child. Uniqueness principle allows to load a class exactly once, which is basically achieved by delegation and ensures that child ClassLoader doesn't reload the class already loaded by parent. Correct understanding of class loader is must to resolve issues like NoClassDefFoundError in Java and java.lang.ClassNotFoundException, which are related to class loading. ClassLoader is also an important topic in advanced Java Interviews, where good knowledge of working of Java ClassLoader and How classpath works in Java is expected from Java programmer. I have always seen questions like, Can one class be loaded by two different ClassLoader in Java on various Java Interviews. In this Java programming tutorial, we will learn what is ClassLoader in Java, How ClassLoader works in Java and some specifics about Java ClassLoader.

27.1 What is ClassLoader

ClassLoader in Java is a class which is used to load class files in Java. Java code is compiled into class file by javac compiler and JVM executes Java program, by executing byte codes written in class file. ClassLoader is responsible for loading class files from file system, network or any other source. There are three default class loader used in Java, Bootstrap , Extension and System or Application class loader.

Every class loader has a predefined location, from where they loads class files. Bootstrap ClassLoader is responsible for loading standard JDK class files from rt.jar and it is parent of all class loaders in Java. Bootstrap class loader don't have any parents, if you call String.class.getClassLoader() it will return null and any code based on that may throw NullPointerException in Java. Bootstrap class loader is also known as Primordial ClassLoader in Java.

Extension ClassLoader delegates class loading request to its parent, Bootstrap and if unsuccessful, loads class form jre/lib/ext directory or any other directory pointed by java.ext.dirs system property. Extension ClassLoader in JVM is implemented by sun.misc.Launcher$ExtClassLoader.

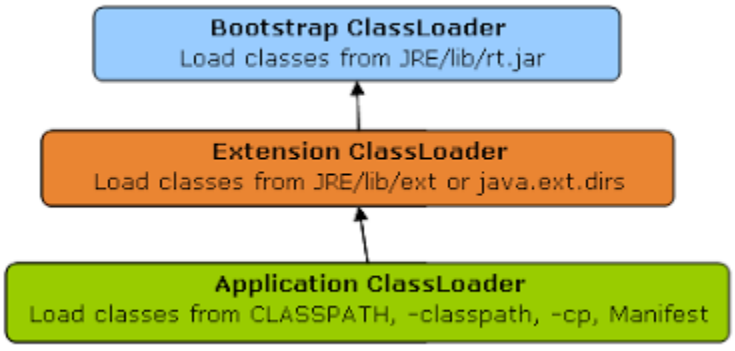
Third default class loader used by JVM to load Java classes is called System or Application class loader and it is responsible for loading application specific classes from CLASSPATH environment variable, -classpath or -cp command line option, Class-Path attribute of Manifest file inside JAR. Application class loader is a child of Extension ClassLoader and its implemented by sun.misc.Launcher$AppClassLoader class. Also, except Bootstrap class loader, which is implemented in native language mostly in C, all Java class loaders are implemented using java.lang.ClassLoader.

In short here is the location from which Bootstrap, Extension and Application ClassLoader load Class files.

1) Bootstrap ClassLoader - JRE/lib/rt.jar

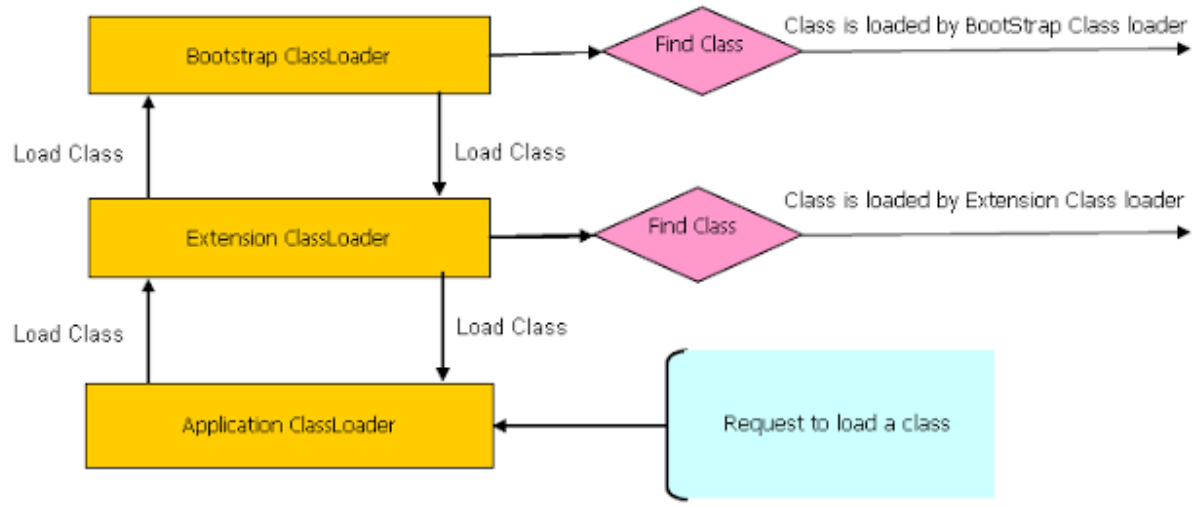
2) Extension ClassLoader - JRE/lib/ext or any directory denoted by java.ext.dirs

3) Application ClassLoader - CLASSPATH environment variable, -classpath or -cp option, Class-Path attribute of Manifest inside JAR file.



27.2 How ClassLoader works

As I explained earlier Java ClassLoader works in three principles : delegation, visibility and uniqueness. In this section we will see those rules in detail and understand working of Java ClassLoader with example. By the way here is a diagram which explains How ClassLoader load class in Java using delegation.



27.3 Delegation principles

a class is loaded in Java, when its needed. Suppose you have an application specific class called Abc.class, first request of loading this class will come to Application ClassLoader which will delegate to its parent Extension ClassLoader which further delegates to Primordial or Bootstrap class loader. Primordial will look for that class in rt.jar and since that class is not there, request comes to Extension class loader which looks on jre/lib/ext directory and tries to locate this class there, if class is found there than Extension class loader will load that class and Application class loader will never load that class but if its not loaded by extension class-loader than Application class loader loads it from Classpath in Java. Remember Classpath is used to load class files while PATH is used to locate executable like javac or java command.

27.4 Visibility Principle

According to visibility principle, Child ClassLoader can see class loaded by Parent ClassLoader but vice-versa is not true. Which mean if class Abc is loaded by Application class loader than trying to load class ABC explicitly using extension ClassLoader will throw either java.lang.ClassNotFoundException. as shown in below Example

```
package test;

import java.util.logging.Level;
import java.util.logging.Logger;

/**
 * Java program to demonstrate How ClassLoader works in Java,
 * in particular about visibility principle of ClassLoader.
 *
 * @author Javin Paul
 */

public class ClassLoaderTest {
```

```
    public static void main(String args[]) {
        try {
            //printing ClassLoader of this class

System.out.println("ClassLoaderTest.getClass().getClassLoader() : "
                                    +
ClassLoaderTest.class.getClassLoader());


            //trying to explicitly load this class again using
Extension class loader
            Class.forName("test.ClassLoaderTest", true
                                ,
ClassLoaderTest.class.getClassLoader().getParent());
        } catch (ClassNotFoundException ex) {

Logger.getLogger(ClassLoaderTest.class.getName()).log(Level.SEVERE,
null, ex);
        }
    }

}

Output:
ClassLoaderTest.getClass().getClassLoader() :
sun.misc.Launcher$AppClassLoader@601bb1
16/08/2012 2:43:48 AM test.ClassLoaderTest main
SEVERE: null
java.lang.ClassNotFoundException: test.ClassLoaderTest
        at java.net.URLClassLoader$1.run(URLClassLoader.java:202)
        at java.security.AccessController.doPrivileged(Native Method)
        at java.net.URLClassLoader.findClass(URLClassLoader.java:190)
        at
sun.misc.Launcher$ExtClassLoader.findClass(Launcher.java:229)
        at java.lang.ClassLoader.loadClass(ClassLoader.java:306)
        at java.lang.ClassLoader.loadClass(ClassLoader.java:247)
        at java.lang.Class.forName0(Native Method)
        at java.lang.Class.forName(Class.java:247)
        at test.ClassLoaderTest.main(ClassLoaderTest.java:29)
```

27.5 Visibility Principle

According to this principle a class loaded by Parent should not be loaded by Child ClassLoader again. Though its completely possible to write class loader which violates Delegation and Uniqueness principles and loads class by itself, its not something which is beneficial. You should follow all class loader principle while writing your own ClassLoader.

27.6 How to load class explicitly

Java provides API to explicitly load a class by Class.forName(classname) and Class.forName(classname, initialized, classloader), remember JDBC code which is used to load JDBC drives we have seen in Java

program to Connect Oracle database. As shown in above example you can pass name of ClassLoader which should be used to load that particular class along with binary name of class. Class is loaded by calling loadClass() method of java.lang.ClassLoader class which calls findClass() method to locate bytecodes for corresponding class. In this example Extension ClassLoader uses java.net.URLClassLoader which search for class files and resources in JAR and directories. any search path which is ended using "/" is considered directory. If findClass() does not found the class than it throws java.lang.ClassNotFoundException and if it finds it calls defineClass() to convert bytecodes into a .class instance which is returned to the caller.

27.7 Where to use ClassLoader

ClassLoader in Java is a powerful concept and used at many places. One of the popular example of ClassLoader is AppletClassLoader which is used to load class by Applet, since Applets are mostly loaded from internet rather than local file system, By using separate ClassLoader you can also loads same class from multiple sources and they will be treated as different class in JVM. J2EE uses multiple class loaders to load class from different location like classes from WAR file will be loaded by Web-app ClassLoader while classes bundled in EJB-JAR is loaded by another class loader. Some web server also supports hot deploy functionality which is implemented using ClassLoader. You can also use ClassLoader to load classes from database or any other persistent store.

28. When a class is loaded and initialized in JVM

Class loading is done by ClassLoaders in Java which can be implemented to eagerly load a class as soon as another class references it or lazy load the class until a need of class initialization occurs. If Class is loaded before its actually being used it can sit inside before being initialized. I believe this may vary from JVM to JVM. While its guaranteed by JLS that a class will be loaded when there is a need of static initialization.

28.1 When Class is loaded

Class loading is done by ClassLoaders in Java which can be implemented to eagerly load a class as soon as another class references it or lazy load the class until a need of class initialization occurs. If Class is loaded before its actually being used it can sit inside before being initialized. I believe this may vary from JVM to JVM. While its guaranteed by JLS that a class will be loaded when there is a need of static initialization.

28.2 When a Class is initialized

After class loading, initialization of class takes place which means initializing all static members of class. A Class is initialized in Java when :

1) An Instance of class is created using either new() keyword or using reflection using class.forName(), which may throw ClassNotFoundException in Java.

2) An static method of Class is invoked.

3) An static field of Class is assigned.

4) An static field of class is used which is not a constant variable.

5) if Class is a top level class and an assert statement lexically nested within class is executed.

Reflection can also cause initialization of class. Some methods of java.lang.reflect package may cause class to be initialized. JLS Strictly says that a class should not be initialized by any reason other than above.

28.3 How Class is initialized

Now we know what triggers initialization of a class in Java, which is precisely documented in Java language specification. Its also important to know in which order various fields (static and non static), block (static an non static), various classes (sub class and super class) and various interfaces (sub interface, implementation class and super interface) is initialized in Java. Infact many Core Java interview question and SCJP question based on this concept because it affect final value of any variable if its initialized on multiple places. Here are some of the rules of class initialization in Java:

1) Classes are initialized from top to bottom so field declared on top initialized before field declared in bottom

2) Super Class is initialized before Sub Class or derived class in Java

3) If Class initialization is triggered due to access of static field, only Class which has declared static field is initialized and it doesn't trigger initialization of super class or sub class even if static field is referenced by Type of Sub Class, Sub Interface or by implementation class of interface.

4) interface initialization in Java doesn't cause super interfaces to be initialized.

5) static fields are initialized during static initialization of class while non static fields are initialized when instance of class is created. It means static fields are initialized before non static fields in Java.

6)non static fields are initialized by constructors in Java. sub class constructor implicitly call super class constructor before doing any initialization, which guarantees that non static or instance variables of super class is initialized before sub class.

28.4 Examples of class initialization

Here is an example of when class is initialized in Java. In this example we will see which classes are initialized in Java.

```
/**
 * Java program to demonstrate class loading and initialization in
Java.
 */
public class ClassInitializationTest {

    public static void main(String args[]) throws InterruptedException
{

        NotUsed o = null; //this class is not used, should not be
initialized
        Child t = new Child(); //initializing sub class, should
trigger super class initialization
        System.out.println((Object)o == (Object)t);
```

```
    }
}

/**
 * Super class to demonstrate that Super class is loaded and
initialized before Subclass.
 */
class Parent {
    static { System.out.println("static block of Super class is
initialized"); }
    {System.out.println("non static blocks in super class is
initialized");}
}

/**
 * Java class which is not used in this program, consequently not
loaded by JVM
 */
class NotUsed {
    static { System.out.println("NotUsed Class is initialized "); }
}

/**
 * Sub class of Parent, demonstrate when exactly sub class loading and
initialization occurs.
 */
class Child extends Parent {
    static { System.out.println("static block of Sub class is
initialized in Java "); }
    {System.out.println("non static blocks in sub class is
initialized");}
}

Output:
static block of Super class is initialized
static block of Sub class is initialized in Java
non static blocks in super class is initialized
non static blocks in sub class is initialized
false
```

28.5 Observation

1) Super class is initialized before sub class in Java.

2) Static variables or blocks are initialized before non static blocks or fields.

3) Not used class is not initialized at all because its not been used, none of the cases mentioned on JLS or above which triggers initialization of class is not happened here.

Let's have a look on another example of class initialization in Java:

```
/**
 * Another Java program example to demonstrate class initialization
and loading in Java.
```

```
 */

public class ClassInitializationTest {

    public static void main(String args[]) throws InterruptedException
{

        //accessing static field of Parent through child, should only
initialize Parent
        System.out.println(Child.familyName);
    }
}

class Parent {
    //compile time constant, accessing this will not trigger class
initialization
    //protected static final String familyName = "Lawson";

    protected static String familyName = "Lawson";

    static { System.out.println("static block of Super class is
initialized"); }
    {System.out.println("non static blocks in super class is
initialized");}
}

Output:
static block of Super class is initialized
Lawson
```

Observation

1. Here class initialization occurs because static field is accessed which is not a compile time constant. had you declare "familyName" compile time constant using final keyword in Java (as shown in commented section) class initialization of super class would not have occurred.

2) Only super class is initialized even though static field is referenced using sub type.

There is another example of class initialization related to interface on JLS which explains clearly that initialization of sub interfaces does not trigger initialization of super interface. I highly recommend reading JLS 14.4 for understating class loading and initialization in more detail.

29. Difference between Comparable and Comparator

While sorting elements in collection comparable and comparator interfaces come into picture which are used to sort collection elements. A natural question which comes to mind is why two different interfaces?

So in this post I'll try to explain the difference between comparable and comparator interfaces and why both of them are required. Before going into the differences between these two, let's have a brief introduction of both.

29.1 Comparable interface

Class whose objects are to be sorted implements this interface. The ordering imposed by the implementation of this interface is referred to as the class's natural ordering, and the class's compareTo method is referred to as its natural comparison method. Classes implementing this interface can be sorted automatically by Collections.sort (and Arrays.sort). Objects that implement this interface can be used as keys in a sorted map (like TreeMap) or as elements in a sorted set (like TreeSet), without the need to specify a comparator.

General form of Comparable interface -

```
public interface Comparable<T> {
 public int compareTo(T o);
}
```

29.2 Comparator interface

In the case of Comparator, class that is to be sorted doesn't implement it. Comparator can be implemented by some other class, as an anonymous class or as lambda expression (from Java 8). Comparators can be passed to a sort method (such as Collections.sort or Arrays.sort) to allow precise control over the sort order. Comparators can also be used to control the order of certain data structures (such as sorted sets or sorted maps).

General form of Comparator interface - Note the annotation @FunctionalInterface, it's available from Java 8 and Comparator being a functional interface can be implemented using lambda expression. Also note that apart from compare method which is the single abstract method there are several other default and static methods too with in the Comparator interface.

```
@FunctionalInterface
public interface Comparator<T> {
  int compare(T o1, T o2);
}
```

29.3 Example code

Let's see an example to further clarify when do we need comparable and when Comparator is needed.

Let's say we have an Employee class. Objects of this Employee class are stored in an array list and we want to sort it first on first name and then on last name. So this order is the natural order for the Employee class, employees are first sorted on first name and then last name.

Employee class

```
public class Employee implements Comparable<Employee>  {
    private String lastName;
    private String firstName;
    private String empId;
    private int age;
```

```
    public String getLastName() {
        return lastName;
    }
    public void setLastName(String lastName) {
        this.lastName = lastName;
    }
    public String getFirstName() {
        return firstName;
    }
    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }
    public String getEmpId() {
        return empId;
    }
    public void setEmpId(String empId) {
        this.empId = empId;
    }
    public int getAge() {
        return age;
    }
    public void setAge(int age) {
        this.age = age;
    }

    @Override
    public String toString() {

        return getFirstName() + " " + getLastName() + " " + getAge() +
" " + getEmpId();
    }
    @Override
    public int compareTo(Employee o) {
        int firstCmp =
this.getFirstName().compareTo(o.getFirstName());
        return firstCmp != 0 ? firstCmp :
this.getLastName().compareTo(o.getLastName());
    }
}
```

Here note that Employee class is implementing Comparable interface providing implementation for the compareTo() method.

Class where sorting of the list will be done

```
public class SortObjectList {
    public static void main(String[] args) {
        List<Employee> empList = new ArrayList<Employee>();
        // Storing elements in the arraylist
        empList.add(getData("E001", "Mishra", "Pyaremohan", 35));
        empList.add(getData("E002", "Smith", "John", 45));
        empList.add(getData("E003", "Sharma", "Ram", 23));
        empList.add(getData("E004", "Mishra", "Pyaremohan", 60));
        empList.add(getData("E005", "Caroll", "Eva", 32));
        empList.add(getData("E003", "Tiwari", "Ram", 23));
```

```
        System.out.println("Original List");
        for(Employee emp : empList){
            System.out.println("" + emp);
        }
        // Sorting the list
        Collections.sort(empList);

        System.out.println("Sorted List");
        for(Employee emp : empList){
            System.out.println("" + emp);
        }
    }

    // Stub method
    private static Employee getData(String empId, String lastName,
String firstName, int age){
        Employee employee = new Employee();
        employee.setEmpId(empId);
        employee.setLastName(lastName);
        employee.setFirstName(firstName);
        employee.setAge(age);
        return employee;
    }
}
Output

Original List
Pyaremohan Mishra 35 E001
John Smith 45 E002
Ram Sharma 23 E003
Pyaremohan Mishra 60 E004
Eva Caroll 32 E005
Ram Tiwari 23 E003
Sorted List
Eva Caroll 32 E005
John Smith 45 E002
Pyaremohan Mishra 35 E001
Pyaremohan Mishra 60 E004
Ram Sharma 23 E003
Ram Tiwari 23 E003
```

29.4 When do we need Comparator

Now in case we want to sort in an order where, if names are same, they are sorted on the basis of age in descending order we can't use the already implemented compareTo() method of the Employee class. So you can say that if we want any ordering other than the defined natural ordering for the class then we have to use Comparator. In that case Comparator will provide an object that encapsulates an ordering.

Sorting logic

```
public class SortObjectList {
    public static void main(String[] args) {
        List<Employee> empList = new ArrayList<Employee>();
        // Storing elements in the arraylist
        empList.add(getData("E001", "Mishra", "Pyaremohan", 35));
        empList.add(getData("E002", "Smith", "John", 45));
        empList.add(getData("E003", "Sharma", "Ram", 23));
        empList.add(getData("E004", "Mishra", "Pyaremohan", 60));
        empList.add(getData("E005", "Caroll", "Eva", 32));
        empList.add(getData("E003", "Tiwari", "Ram", 23));

        System.out.println("Original List");
        for(Employee emp : empList){
            System.out.println("" + emp);
        }
        // Sorting the list
        Collections.sort(empList, new MyComparator());

        System.out.println("Sorted List");
        for(Employee emp : empList){
            System.out.println("" + emp);
        }
    }

    // Stub method
    private static Employee getData(String empId, String lastName,
String firstName, int age){
        Employee employee = new Employee();
        employee.setEmpId(empId);
        employee.setLastName(lastName);
        employee.setFirstName(firstName);
        employee.setAge(age);
        return employee;
    }

}

class MyComparator implements Comparator<Employee>{
    @Override
    public int compare(Employee o1, Employee o2) {
        int firstCmp = o1.getFirstName().compareTo(o2.getFirstName());
        if(firstCmp == 0){
            int lastCmp =
o1.getLastName().compareTo(o2.getLastName());
            if(lastCmp == 0){
                return (o2.getAge() < o1.getAge() ? -1 :
                        (o2.getAge() == o1.getAge() ? 0 : 1));
            }else{
                return lastCmp;
            }

        }else{
            return firstCmp;
        }
```

```
    }
}
Output

Original List
Pyaremohan Mishra 35 E001
John Smith 45 E002
Ram Sharma 23 E003
Pyaremohan Mishra 60 E004
Eva Caroll 32 E005
Ram Tiwari 23 E003
Sorted List
Eva Caroll 32 E005
John Smith 45 E002
Pyaremohan Mishra 60 E004
Pyaremohan Mishra 35 E001
Ram Sharma 23 E003
Ram Tiwari 23 E003
```

Here it can be seen that the name which is same is sorted by age in descending order.

Note here that a Comparator class MyComparator is implementing the Comparator interface and providing implementation for the compare() method. Also note that in Collections.sort method now we need to provide the Comparator class.

```
Collections.sort(empList, new MyComparator());
```

Comparator class can also be implemented as an Anonymous class or as a Lambda expression.

Comparable	**Comparator**
Comparable interface is in **java.lang** package.	**Comparator** interface is in **java.util** package.
Comparable interface provides **public int compareTo(T o);** method which needs to be implemented for sorting the elements. This method compares this object with object o and returns an integer, if that integer is - • **Positive** - this object is greater than o. • **Zero** - this object is equal to o. • **Negative** - this object is less than o.	**Comparator** interface provides **int compare(T o1, T o2);** method which needs to be implemented for sorting the elements. Here o1 and o2 objects are compared and an integer value is returned, if that integer is - • **Positive** - o1 is greater than o2. • **Zero** - o1 is equal to o2. • **Negative** - o1 is less than o2.
The class which has to be sorted should implement the **comparable** interface (sorting logic is in the class which has to	Some other class can implement the **Comparator** interface not the actual class whose objects are to be sorted. That way there may be many comparators and depending on

Comparable	Comparator
be sorted), and that implementation becomes the natural ordering of the class.	the ordering needed specific comparator can be used. As example for Employee class if different orderings are needed based on first name, last name, age etc. then we can have different comparators with the implementations for those orderings.
To sort the list when **Comparable** is used we can use **Collections.sort(List)**.	To sort the list when **Comparator** is used, Comparator class has to be explicitly specified as a param in the sort method.**Collections.sort(List, Comparator)**
A comparable object is capable of comparing itself with another object. The class itself must implements the java.lang.Comparable interface in order to be able to compare its instances.	A comparator object is capable of comparing two different objects. The class is not comparing its instances, but some other class's instances. This comparator class must implement the java.util.Comparator interface.
Use Comparable if you want to define a default (natural) ordering behaviour of the object in question, a common practice is to use a technical or natural (database?) identifier of the object for this.	Use Comparator if you want to define an external controllable ordering behaviour, this can override the default ordering behaviour.

29.5 When to use Comparator and Comparable

1) If there is a natural or default way of sorting Object already exist during development of Class than use Comparable. This is intuitive and you given the class name people should be able to guess it correctly like Strings are sorted chronically, Employee can be sorted by there Id etc. On the other hand if an Object can be sorted on multiple ways and client is specifying on which parameter sorting should take place than use Comparator interface. for example Employee can again be sorted on name, salary or department and clients needs an API to do that. Comparator implementation can sort out this problem.

2) Some time you write code to sort object of a class for which you are not the original author, or you don't have access to code. In these cases you can not implement Comparable and Comparator is only way to sort those objects.

3) Beware with the fact that How those object will behave if stored in SorteSet or SortedMap like TreeSet and TreeMap. If an object doesn't implement Comparable than while putting them into SortedMap, always provided corresponding Comparator which can provide sorting logic.

4) Order of comparison is very important while implementing Comparable or Comparator interface. for example if you are sorting object based upon name than you can compare first name or last name on any order, so decide it judiciously. I have shared more detailed tips on compareTo on my post how to implement CompareTo in Java.

5) Comparator has a distinct advantage of being self descriptive for example if you are writing Comparator to compare two Employees based upon there salary than name that comparator as SalaryComparator, on the other hand compareTo().

30. Final Vs finally Vs finalize

30.1 final

final keyword is used to restrict in some way. It can be used with variables, methods and classes. When a variable is declared as final, its value can not be changed once it is initialized. Except in case of blank final variable, which must be initialized in the constructor. If you make a method final in Java, that method can't be overridden in a sub class.If a class is declared as final then it can not be sub classed.

final keyword in java has its usage in preventing the user from modifying a field, method or class.

1. final field - A variable declared as final prevents the content of that variable being modified.
2. final method - A method declared as final prevents the user from overriding that method.
3. final class - A class declared as final can not be extended thus prevents inheritance.

30.1.1 final variable in Java

A variable can be declared as final which prevents its contents from being modified. A final variable can be initialized only once but it can be done in two ways.

Value is assigned when the variable is declared.

Value is assigned with in a constructor.

Please note that the final variable that has not been assigned a value while declaring a variable is called blank final variable, in that case it forces the constructor to initialize it.

30.1.2 Final object

Making an object reference variable final is a little different from a normal variable as in that case object fields can still be changed but the reference can't be changed. That is applicable to collections too, a collection declared as final means only its reference can't be changed values can still be added, deleted or modified in that collection.

30.1.3 final method parameter

parameters of a method can also be declared as final to make sure that parameter value is not changed in the method. Since java is pass by value so changing the parameter value won't affect the original value anyway. But making the parameter final ensures that the passed parameter value is not changed in the method and it is just used in the business logic the way it needs to be used.

30.1.4 Final variables and thread safety

Variables declared as final are essentially read-only thus thread safe.

Also if any field is final it is ensured by the JVM that other threads can't access the partially constructed object, thus making sure that final fields always have correct values when the object is available to other threads.

30.1.5 final with inheritance

The use of final with the method or class apply to the concept of inheritance.

30.1.6 final method

A method can be declared as final in order to avoid method overriding. Method declared as final in super class cannot be overridden in subclass. If creator of the class is sure that the functionality provided in a method is complete and should be used as is by the sub classes then the method should be declared as final.

30.1.7 Benefit of final method

Methods declared as final may provide performance enhancement too. Generally in Java, calls to method are resolved at run time which is known as late binding.

Where as in case of final methods since compiler knows these methods can not be overridden, call to these methods can be resolved at compile time. This is known as early binding. Because of this early binding compiler can inline the methods declared as final thus avoiding the overhead of method call.

30.1.8 Final Class

A class declared as final can't be extended thus avoiding inheritance altogether.

If creator of the class is sure that the class has all the required functionality and should be used as it is with out extending it then it should be declared as final.

Declaring a class as final implicitly means that all the methods of the class are final too as the class can't be extended.

It is illegal to declare a class as both final and abstract. Since abstract class by design relies on subclass to provide complete implementation.

30.1.9 Benefits of final class

Since all the methods in a final class are implicitly final thus final class may provide the same performance optimization benefit as final methods.

Restricting extensibility may also be required in some cases essentially when using third party tools.

A class may be declared final to enforce immutability.

30.1.10 Points to note

- final keyword can be used with fields, methods and classes.

- final variables are essentially read only and it is a common convention to write the final variables in all caps. As exp. Private final int MAXIMUM_VALUE = 10
- final variables that are not initialized during declaration are called blank final variable in that case it forces the constructor to initialize it. Failure to do so will result in compile time error.
- final variables can be initialized only once, assigning any value there after will result in compile time error.
- final variables are read-only thus thread safe.
- In case of Final object reference can't be changed.
- final methods can't be overridden
- final classes can't be extended.
- final methods are bound during compile time (early binding), which means compiler may inline those methods resulting in performance optimization.

30.2 finally

finally is part of exception handling mechanism in Java. finally block is used with try-catch block. Along with a try block we can have both catch and finally blocks or any one of them. So we can have any of these combinations try-catch-finally, try-catch, try-finally. finally block is always executed whether any exception is thrown or not and raised exception is handled in catch block or not. Since finally block always executes thus it is primarily used to close the opened resources like database connection, file handles etc.

From Java 7 onwards try with resources provide another way to automatically manage resources.

When an exception occurs in the code, the flow of the execution may change or even end abruptly. That may cause problem if some resources were opened in the method.

As exp if a file was opened in a method and it was not closed in the end as some exception occurred then the resources may remain open consuming memory. finally provides that exception-handling mechanism to clean up.

Code with in the finally block will be executed after a try/catch block has completed. The finally block will be executed whether or not an exception is thrown.

If the code with in try block executes without throwing any exception, the finally block is executed immediately after the completion of try block. If an exception is thrown with in the try block, the finally block is executed immediately after executing the catch block which handled the thrown exception.

30.2.1 Example of finally

Even if there is a return statement in try block finally will be executed.

```
public class FinallyDemo {

 public static void main(String[] args) {
  int i = displayValue();
  System.out.println("Value of i " + i);

 }

 static int displayValue(){
```

```
  try{
   System.out.println("In try block");
   return 1;
  }finally{
   System.out.println("In finally block");
   return 2;
  }
 }

}
Output

In try block
In finally block
Value of i 2
```

It can be seen though try block has a return value still finally is executed. Ultimately return value is what finally retruns.

Even if the exception is thrown still return value will be what finally returns. Let's see the same example again, this time an exception will be thrown.

```
public class FinallyDemo {

 public static void main(String[] args) {
  int i = displayValue();
  System.out.println("Value of i " + i);

 }

 static int displayValue(){
  try{
   System.out.println("In try block");
   throw new NullPointerException();
  }catch(NullPointerException nExp){
   System.out.println("Exception caught in catch block of
displayValue");
   return 2;
  }finally{
   System.out.println("In finally block");
   //return 3;
  }
 }

}
Output

In try block
Exception caught in catch block of displayValue
In finally block
Value of i 3
```

Please note that it is not a good idea to return any thing from finally as it will suppress the exception. finally must be used for cleaning up.

30.2.2 Points to note

- Every try block must be followed by at least one catch or a finally clause. So we can have any of these combinations try-catch, try-finally or try-catch-finally blocks.
- When method is about to return from its try-catch block after normal execution flow or because of an uncaught exception, finally block is executed just before the method returns.
- If there are multiple catch blocks and none of them can handle the exception thrown, even then the finally bock is executed.
- Though there can be multiple catch blocks associated with a single try statement but there can only be single finally block associated with a try block. In case of nested try blocks there can be associated finally blocks with respective try blocks.
- Finally block is used to close the allocated resources (like file handles, database connections) before returning from the method.
- It's not a good idea to return any value from finally as it may cause any exceptions that are not caught in the method to be lost.
- If the JVM exits (through System.exit() or JVM crash) while the try or catch code is being executed, then the finally block may not execute. In case of multithreading if the thread executing the try or catch code is interrupted or killed, the finally block may not execute even though the application as a whole continues.

30.3 finalize()

finalize() method is a protected method of java.lang.Object class. Since it is in Object class thus it is inherited by every class. This method is called by garbage collector thread before removing an object from the memory. This method can be overridden by a class to provide any cleanup operation and gives object final chance to cleanup before getting garbage collected.

```
protected void finalize() throws Throwable
{
    //resource clean up operations
}
```

Please note that it is not a good idea to rely on finalize() method for closing resources as there is no guarantee when finalize() method will be called by Garbage collector.

There may be a situation when an object will need to perform some action just before it is getting garbage collected. For example if an object is holding some non-java resources like file handle then it is better to make sure that these resources are closed before an object is destroyed because simply reclaiming the memory used by an object would not guarantee that the resources it held would be reclaimed.

For that purpose Java provides a mechanism called finalization through finalize() method. In finalize method we can provide the actions to release the resources before the object is destroyed.

30.3.1 When is finalize() method called

As we know that in Java object memory deallocation happens automatically through garbage collection. In Java, garbage collector is run by the run time environment periodically and it looks for the objects for which there are no existing references. finalize method for an object is executed just before it is garbage collected. But here lies the caveat According to Java language specification - http://docs.oracle.com/javase/specs/jls/se7/html/jls-12.html#jls-12.6 The Java programming language

does not specify how soon a finalizer will be invoked that's one reason why we should not rely solely on finalize() method to release resources.

According to Joshua Bloch in his book Effective Java, Item 7: Avoid Finalizers

"The promptness with which finalizers are executed is primarily a function of the garbage collection algorithm, which varies widely from JVM implementation to JVM implementation. The behavior of a program that depends on the promptness of finalizer execution may likewise vary. It is entirely possible that such a program will run perfectly on the JVM on which you test it and then fail miserably on the JVM favored by your most important customer."

30.3.2 General form of finalize method

finalize method is provided as a protected method in the Object class.

protected void finalize() throws Throwable

Just as a reminder the protected access modifier provides the access as - Variables, methods and constructors which are declared protected in a superclass can be accessed only by the subclasses in other package or any class within the package of the protected members' class.

Since all the classes have Object class as the super class either directly or indirectly so this method would be accessible to each and every class anyway so it makes sense to keep the access as protected.

30.3.3 How to use finalize method

The finalize method of class Object performs no special action; it simply returns normally. Any class which needs some functionality to be provided in finalize() method has to override this definition and provide the functionality. According to Java docs - The finalize method may take any action, including making this object available again to other threads; the usual purpose of finalize, however, is to perform cleanup actions before the object is irrevocably discarded.

30.3.4 How to properly write a finalize() method

Like constructor chaining in Java there is no finalizer chaining so super class finalize method would not be called automatically. As example If there is class A which provides a finalize method, if Class A is extended by class B and class B overrides the finalize method then only Class B's finalize method will be called. Class A's finalize method would not be called automatically. Thus we need to explicitly call it using super -

```
protected void finalize() throws Throwable {
     try {
         // clean up
     } finally {
         super.finalize(); // call parent class' finalize
     }
}
```

30.3.5 Example code for finalize method

Here is a simple example where I have created an object then set that object reference as null. Also called System.gc() to run garbage collector. Note you may not get the output in some of the runs as System.gc() is also more of a suggestion to the virtual machine to reclaim memory from unused objects it may not run as soon as System.gc() is called and the program may terminate not at all calling the finalize method.

```
public class FinalizeDemo {
    int i;
    FinalizeDemo(int num){
        this.i = num;
    }
    public static void main(String[] args) {
        // creating object
        FinalizeDemo finalizeDemo = new FinalizeDemo(10);
        Temp temp = new Temp();
        // setting object reference as null so it is
        // eligible for garabge collection
        finalizeDemo = null;
        temp.doCalcualtion();
        // setting object reference as null so it is
        // eligible for garabge collection
        temp = null;
        // Calling System.gc() to run garbage collector
        System.gc();
    }

    @Override
    protected void finalize() throws Throwable {
        try{
            System.out.println("finalize method called for
FinalizeDemo");
        }finally{
            super.finalize();
        }
    }

}

class Temp{
    public void doCalcualtion(){
        int i = 5;
        System.out.println("value of i is " + i);
    }
    @Override
    protected void finalize() throws Throwable {
        try{
            System.out.println("finalize method called for Temp");
        }finally{
            super.finalize();
        }
    }
```

```
}
Output

value of i is 5
finalize method called for Temp
finalize method called for FinalizeDemo
```

Though I have used System.gc here to some how make sure that finalize() method is indeed called but don't rely on it. To quote Joshua Bloch again "Don't be seduced by the methods System.gc and System.runFinalization. They may increase the odds of finalizers getting executed, but they don't guarantee it. The only methods that claim to guarantee finalization are System.runFinalizersOnExit and its evil twin, Runtime.runFinalizersOnExit. These methods are fatally flawed and have been deprecated."

30.3.6 When we can use finalize method

Though there are other better alternatives to clean up resources like a finally block or try-with-resources(ARM) available from Java 7. But to make sure, just as an extra safety measure, that resources are indeed closed before the object is destroyed finalize() method may be used. But be forewarned it does comes with a hit on performance.

30.3.7 Finalize method and exception handling

Any exception thrown by the finalize method while finalize method is executing causes the finalization of this object to be halted, but is otherwise ignored.

30.3.8 Points to note

- Every class inherits the finalize() method from java.lang.Object.
- The finalize method is called by the garbage collector when it determines no more references to the object exist
- The finalize method of class Object performs no special action; it simply returns normally. Subclasses of Object may override this definition.
- If overridding finalize() it is a good programming practice to use a try-catch-finally statement and to always call super.finalize() as there is no concept of finalizer chaining.
- The finalize method is never invoked more than once by a Java virtual machine for any given object.

31. fail-fast Vs fail-safe iterator

The collections which are there from Java 1.2 (or even legacy) like ArrayList, Vector, HashSet all have fail-fast iterator whereas Concurrent collections added in Java 1.5 like ConcurrrentHashMap, CopyOnWriteArrayList, CopyOnWriteArraySet have fail-safe iterator.

So first let us see the key differences between the fail-fast and fail-safe iterator and then we'll see some programs to explain those features.

1. fail-fast iterator throws ConcurrentModificationException if the underlying collection is structurally modified in any way except through the iterator's own remove or add methods.
2. fail-safe iterator doesn't throw ConcurrentModificationException.

fail-fast iterator work on the original collection.

fail-safe iterator makes a copy of the underlying structure and iteration is done over that snapshot. Drawback of using a copy of the collection rather than original collection is that the iterator will not reflect additions, removals, or changes to the collection since the iterator was created.

3. fail-fast iterator provides remove, set, and add operations. Note that not all the iterators support all these methods. As exp. ListIterator supports add() method but the general iterator doesn't.

With fail-safe iterator element-changing operations on iterators themselves (remove, set, and add) are not supported. These methods throw UnsupportedOperationException.

Now let us see some detailed explanation and supporting programs to see these features of both fail-fast and fail-safe iterators.

31.1 fail-fast iterator

An iterator is considered fail-fast if it throws a ConcurrentModificationException under either of the following two conditions:

- In multi-threaded environment, if one thread is trying to modify a Collection while another thread is iterating over it.
- Even with single thread, if a thread modifies a collection directly while it is iterating over the collection with a fail-fast iterator, the iterator will throw this exception.

fail-fast iterator fails if the underlying collection is structurally modified at any time after the iterator is created, thus the iterator will throw a ConcurrentModificationException if the underlying collection is structurally modified in any way except through the iterator's own remove or add (if applicable as in list-iterator) methods.

Note that structural modification is any operation that adds or deletes one or more elements; merely setting the value of an element (in case of list) or changing the value associated with an existing key (in case of map) is not a structural modification.

Mostly Iterators from java.util package throw ConcurrentModificationException if collection was modified by collection's methods (add / remove) while iterating

Also note that according to Oracle Docs fail-fast behavior of an iterator cannot be guaranteed as it is, generally speaking, impossible to make any hard guarantees in the presence of unsynchronized concurrent modification. Fail-fast iterators throw ConcurrentModificationException on a best-effort basis. Therefore, it would be wrong to write a program that depended on this exception for its correctness: the fail-fast behavior of iterators should be used only to detect bugs.

Example code of fail-fast iterator with an attempt to add new value to a map while the map is being iterated

```
public class FailFastModification {
    public static void main(String[] args) {
        // creating map
        Map <String,String> cityMap = new HashMap<String,String>();
        cityMap.put("1","New York City" );
        cityMap.put("2", "New Delhi");
        cityMap.put("3", "Newark");
        cityMap.put("4", "Newport");
```

```
        // getting iterator
        Iterator <String> itr = cityMap.keySet().iterator();
        while (itr.hasNext()){
            System.out.println(cityMap.get(itr.next()));
            // trying to add new value to a map while iterating it
            cityMap.put("5", "New City");
        }
    }

}
OUTPUT

New York City
Exception in thread "main" java.util.ConcurrentModificationException
 at java.util.HashMap$HashIterator.nextNode(Unknown Source)
 at java.util.HashMap$KeyIterator.next(Unknown Source)
 at
org.netjs.examples.FailFastModification.main(FailFastModification.java
:20)
```

Though we can update the underlying collection, let's see an example with the same hash map used above -

```
public class FailFastModification {
    public static void main(String[] args) {
        // creating map
        Map <String,String> cityMap = new HashMap<String,String>();
        cityMap.put("1","New York City" );
        cityMap.put("2", "New Delhi");
        cityMap.put("3", "Newark");
        cityMap.put("4", "Newport");
        // getting iterator
        Iterator <String> itr = cityMap.keySet().iterator();
        while (itr.hasNext()){
            System.out.println(cityMap.get(itr.next()));
            // updating existing value while iterating
            cityMap.put("3", "New City");
        }
    }
}
Here I have changed the value for the key "3", which is reflected in
the output and no exception is thrown.

New York City
New Delhi
New City
Newport
Using iterator remove method I can remove the values, that is
permitted.

public class FailFastModification {
    public static void main(String[] args) {
        Map <String,String> cityMap = new HashMap<String,String>();
```

```
        cityMap.put("1","New York City" );
        cityMap.put("2", "New Delhi");
        cityMap.put("3", "Newark");
        cityMap.put("4", "Newport");
        System.out.println("size before iteration " + cityMap.size());
        Iterator <String> itr = cityMap.keySet().iterator();
        while (itr.hasNext()){
            System.out.println(cityMap.get(itr.next()));
            // removing value using iterator remove method
            itr.remove();
        }
        System.out.println("size after iteration " + cityMap.size());
    }
}
Output

size before iteration 4
New York City
New Delhi
Newark
Newport
size after iteration 0
```

Here after iteration the value is removed using the remove method of the iteartor, thus the size becomes zero after the iteration is done.

Let's see a multi-threaded example, where concurrency is simulated using sleep method.

```
public class FailFastTest {
 public static void main(String[] args) {
  final Map<String,String> cityMap = new HashMap<String,String>();
  cityMap.put("1","New York City" );
  cityMap.put("2", "New Delhi");
  cityMap.put("3", "Newark");
  cityMap.put("4", "Newport");
  //This thread will print the map values
  // Thread1 starts
  Thread thread1 = new Thread(){
   public void run(){
    try{
     Iterator i = cityMap.keySet().iterator();
     while (i.hasNext()){
      System.out.println(i.next());
      // Using sleep to simulate concurrency
      Thread.sleep(1000);
     }
    }catch(ConcurrentModificationException e){
     System.out.println("thread1 : Concurrent modification detected on
this map");
    }catch(InterruptedException e){

    }
   }
  };
  thread1.start();
```

```
  // Thread1 ends
   // This thread will try to remove value from the collection,
  // while the collection is iterated by another thread.
  // thread2 starts
  Thread thread2 = new Thread(){
   public void run(){
     try{
    // Using sleep to simulate concurrency
      Thread.sleep(2000);
      // removing value from the map
      cityMap.remove("2");
      System.out.println("city with key 2 removed successfully");
     }catch(ConcurrentModificationException e){
      System.out.println("thread2 : Concurrent modification detected
on this map");
     } catch(InterruptedException e){}
    }
  };
  thread2.start();
// thread2 end
 } // main end
} // class end
Output

1
2
city with key 2 removed successfully
thread1 : Concurrent modification detected on this map
It can be seen that in thread 1 which is iterating the map, Concurrent
modification exception is thrown.

If we use the same multi-threaded program for updating the values, it
should update the values.

public class FailFastTest {
 public static void main(String[] args) {
  final Map<String,String> cityMap = new HashMap<String,String>();
  cityMap.put("1","New York City" );
  cityMap.put("2", "New Delhi");
  cityMap.put("3", "Newark");
  cityMap.put("4", "Newport");
  //This thread will print the map values
  Thread thread1 = new Thread(){
   public void run(){
    try{
     Iterator i = cityMap.keySet().iterator();
     while (i.hasNext()){
      System.out.println(cityMap.get(i.next()));
      // Using sleep to simulate concurrency
      Thread.sleep(1000);
     }
    }catch(ConcurrentModificationException e){
     System.out.println("thread1 : Concurrent modification detected on
this map");
```

```
    }catch(InterruptedException e){

    }
   }
  };
  thread1.start();

  Thread thread2 = new Thread(){
   public void run(){
     try{
    // Using sleep to simulate concurrency
      Thread.sleep(500);
      // removing value from the map
      cityMap.put("2", "New City");
      System.out.println("city with key 2 updated successfully");
     }catch(ConcurrentModificationException e){
      System.out.println("thread2 : Concurrent modification detected
on this map");
     } catch(InterruptedException e){}
    }
  };
  thread2.start();
 } // main end
} // class end
Output

New York City
city with key 2 updated successfully
New City
Newark
Newport
```

It can be seen while one thread is iterating through the map another thread is updating one of the value but that doesn't result in concurrentmodificationException.

31.2 Fail Safe iterator

In case of fail-safe iterator, ConcurrentModificationException is not thrown as the fail-safe iterator makes a copy of the underlying structure and iteration is done over that snapshot.

Since iteration is done over a copy of the collection so interference is impossible and the iterator is guaranteed not to throw ConcurrentModificationException.

Drawback of using a copy of the collection rather than original collection is that the iterator will not reflect additions, removals, or changes to the collection since the iterator was created. Element-changing operations on iterators themselves (remove, set, and add) are not supported. These methods throw UnsupportedOperationException.

Iterator of CopyOnWriteArrayList is an example of fail-safe Iterator also iterator provided by ConcurrentHashMap keySet is fail-safe and never throws ConcurrentModificationException.

```
Example code with fail-safe
```

```
public class FailSafeTest {
 public static void main(String[] args) {

  List <String> cityList = new CopyOnWriteArrayList<String>();
  cityList.add("New York City");
  cityList.add("New Delhi");
  cityList.add("Newark");
  cityList.add("Newport");
  Iterator<String> itr = cityList.iterator();
  while (itr.hasNext())
          {
           System.out.println(itr.next());
           cityList.add("NewCity");
           //itr.remove();
          }
 }
}
```

This program won't throw ConcurrentModificationException as iterator used with CopyOnWriteArrayList is fail-safe iterator.

If we uncomment the line //itr.remove(); this program will throw UnsupportedOperationException as fail-safe iterator does not support remove, set, and add operations.

31.3 Points to note

- An iterator is considered fail-fast if it throws a ConcurrentModificationException in case the underlying collection's structure is modified.
- While iterating a list or a map values can be updated, only if an attempt is made to add or remove from the collection ConcurrentModificationException will be thrown by fail-fast iterator.
- Fail-fast iterators throw ConcurrentModificationException on a best-effort basis and fail-fast behavior of an iterator cannot be guaranteed.
- fail-safe iterator works with a copy of the collection rather than the original collection thus interference is impossible and the iterator is guaranteed not to throw ConcurrentModificationException.
- remove, set, and add operations are not supported with fail-safe iterator.

32. What are the changes in Collections framework in Java 8?

- Stream API - Stream can be obtained for Collection via the stream() and parallelStream() methods;
  As exp if you have a list of cities in a List and you want to remove the duplicate cities from that list it can be done as

```
cityList = cityList.stream().distinct().collect(Collectors.toList());
```

- ForEach loop which can be used with Collection. Spliterators which are helpful in parallel processing where several threads can itearate/process part of the collection.
- New methods are added to Collections like replaceAll, getOrDefault, putIfAbsent in Map.
- HashMap, LinkedHashMap and ConcurrentHashMap.implementation is changed to reduce hash collisions. Instead of linked list a balanced tree is used to store the entries after a certain threshold is reached.

33. Overview of lambda expression in Java 8

Lambda expressions (also known as closures) are one of the most important feature added in Java 8. Lambda expressions provide a clear and elegant way to represent a single abstract method interface (Functional interface) using an expression.

Though the true power of lambda expression comes into fore when used with stream API where lambda expressions can be used to provide an easy yet efficient way to perform data manipulation operations like sort, filter, map, reduce etc. But that is a topic for another post! Here I'll just stick to fundamentals of Lambda expressions, functional interface and how lambda expressions can provide implementation of the abstract method defined by the functional interface.

It is very important to have a good understanding of two terms to know lambda expression - lambda expression itself and functional interface.

33.1 What is a lambda expression?

Lambda expression in itself is an anonymous method i.e. a method with no name which is used to provide implementation of the method defined by a functional interface.

A new syntax and operator has been introduced in Java for Lambda expressions.

General form of lambda expression

```
(optional) (Arguments) -> body
```

Here you can see a new arrow operator or lambda operator which divides the lambda expression into two parts. Left side specifies parameters required by the lambda expression. Parameters are optional, if no parameters are needed an empty parenthesis can be given. Right side known as lambda body specifies the logic of the lambda expression.

Let's see some examples of the lambda expressions -

```
() -> 7;
```

This lambda expression takes no parameter, that's why empty parenthesis and returns the constant value 7.

If we have to write the equivalent Java method then it will be something like this -

```
int getValue(){
    return 7;
}
```

Some other examples -

```
// concatenating two strings, it has 2 string parameters and they are
concatenated in lambda body
(String s1, String s2) -> s1+s2;
// Lambda expression to test if the given number is odd or not
n -> n % 2 != 0;
// Prints the passed string s to the console and returns void
```

```
(String s) -> { System.out.println(s); };
```

33.2 What is a functional interface?

A functional interface is an interface with only one abstract method. A functional interface is also known as SAM type where SAM stands for (Single Abstract Method). An example of functional interface would be Runnable interface which has only one method run().

Please note that from Java 8 it is possible for an interface to have default method and static method thus the stress on "only one abstract method".

Also a new annotation @FunctionalInterface has been added in Java 8, all the functional interfaces can be annotated with this annotation to ensure that they have only single abstract method.

Example of functional interface

```
interface IMyInterface {
    int getValue();
}
```

In interface IMyInterface there is only single abstract method getValue() (note that in an interface methods are implicitly abstract).

33.3 Type of Lambda expression

Lambda expression is an instance of the functional interface and provides implementation of the abstract method defined by the functional interface. Thus the functional interface specifies its target type.

Lambda supports "target typing" which infers the object type from the context in which it is used. To infer that object type from the context -

- The parameter type of the abstract method and the parameter type of the lambda expression must be compatible. For Example, if the abstract method in the functional interface specifies one int parameter, then the lambda should also have one int parameter explicitly defined or implicitly inferred as int by the context.
- Its return type must be compatible with the method's type.
- Lambda expression can throw only those exceptions which are acceptable to the method.

Target typing is used in a number of contexts including the following -

- Variable declarations
- Assignments
- Return statements
- Array initializers
- Method or constructor arguments
- Lambda expression bodies

Example of lambda expression

```
interface IMyInterface {
    int getValue();
}
```

```
public class LambdaExpressionDemo {
    public static void main(String[] args) {
        // reference of the interface
        IMyInterface objRef;
       // Lambda expression
       objRef = () -> 7;
       System.out.println("Value is " + objRef.getValue());
       // Another lambda expression using the same interface reference
       objRef = () -> 7 * 5;
       System.out.println("Value is " + objRef.getValue());
       // This line will give compiler error as target type
       // can't be inferred
       objref = () -> "11";
    }
}
Output

Value is 7
Value is 35
```

Here we have an interface IMyInterface with one method getValue() whose return type is int. Since there is only one abstract method in the interface so IMyInterface is a functional interface.

In this program it can be seen how lambda expression () -> 7 is compatible with the abstract method. Return type is int and there is no method parameter.

If you uncomment the line () -> "11"; it will give compiler error The target type of this expression must be a functional interface as in this case return type is string which is not compatible with the return type of the abstract method, lambda expression is implementing.

Also notice that the same functional interface reference is used to execute 2 lambda expressions

objRef = () -> 7; and objRef = () -> 7 * 5;

Since both of the lambda expressions are compatible with the target type so both of them can be assigned to the same reference (Does it remind you of run time polymorphism?). That's why Lambda Expression is a Poly Expression.

Example of lambda expression with a parameter

```
interface IMyInterface {
    boolean test(int n);
}

public class LambdaExpressionDemo {
    public static void main(String[] args) {
        IMyInterface objRef;
        // Lambda expression
        objRef = n -> (n % 2) == 0;
        System.out.println("4 is even " + objRef.test(4));
        System.out.println("3 is even " + objRef.test(3));
    }
}
Output
```

```
4 is even true
3 is even false
```

Here we need to note certain points -

n -> (n % 2) == 0;

In this lambda expression type is not specified explicitly as int, type is inferred from the context in which the lambda expression is executed, though type can be given explicitly too.

int n -> (n % 2) == 0; this lambda expression is also valid.

Also parenthesis around the parameter is omitted, in case of single parameter it is not necessary to enclose the parameter with parenthesis. Again it won't be invalid to do that, so (n) -> (n % 2) == 0; or (int n) -> (n % 2) == 0; are also valid.

In case of more than one parameter also type can be inferred so

(int x, int y) -> x+y; or (x, y) -> x + y;

both of these may be valid if used in a correct context.

One important point here is we can't have lambda expression where type for only one of the parameter is explicitly declared.

// Invalid lambda expression

(int x, y) -> x + y;

Block lambda expression

Till now all the examples we have seen are single expression lambda, we also have a second type of lambda expressions known as "block lambda" where the right side of the lambda expression is a block of code.

Let's see an example, here with in the lambda expression we have a block of code for counting the words in a string without using any String function.

```
@FunctionalInterface
interface IMyInterface {
  int doCount(String str);
}
public class LambdaExpressionDemo {
  public static void main(String[] args) {
    // Lambda expression
    IMyInterface objRef = (str) -> {
      int c = 0;
      char ch[]= new char[str.length()];
      for(int i = 0; i < str.length(); i++){
          ch[i] = str.charAt(i);
          if(((i > 0) && (ch[i] != ' ') && (ch[i-1] == ' ')) ||
          ((ch[0] != ' ') && (i == 0)))
              c++;
      }
      return c;
    };
```

```
    System.out.println("Words in the string " + objRef.doCount("Lambda
Expression in Java"));

  }
}
Output

Words in the string 4
```

Note here that functional interface is annotated with a @FunctionalInterface annotation, this is a new annotation added in Java 8 to be used with functional interface.

33.4 Points to note

- Lambda expression in itself is an anonymous method which implements the method of the functional interface.
- The lambda expression signature must be the same as the functional interface method's signature, as the target type of the lambda expression is inferred from that context.
- A lambda expression can throw only those exceptions or the subtype of the exceptions for which an exception type is declared in the functional interface method's throws clause.
- The parameter list of the lambda expression can declare a vararg parameter: (int ... i) -> {};
- A Lambda Expression Is a Poly Expression - The type of a lambda expression is inferred from the target type thus the same lambda expression could have different types in different contexts. Such an expression is called a poly expression.

34. Difference between iterator and ListIterator?

- Iterator can be obtained on any Collection class like List or Set. But ListIterator can only be used to traverse a List.
- Iterator only moves in one direction using next() method. ListIterator can iterate in both directions using next() and previous() methods.
- Iterator always start at the beginning of the collection.
- ListIterator can be obtained at any point.
  As Exp. If you have a List of integers numberList, you can obtain a ListIterator from the third index of this List.

```
ListIterator<Integer> ltr = numberList.listIterator(3);
```

- ListIterator provides an add(E e) method which is not there in Iterator. add(E e) inserts the specified element into the list.
- ListItearator also provides set method. void set(E e) replaces the last element returned by next() or previous() with the specified element.

35. What happens if HashMap has duplicate keys?

If an attempt is made to add the same key twice, it won't cause any error but the value which is added later will override the previous value.
As Exp. If you have a Map, cityMap and you add two values with same key in the cityMap, Kolkata will override the New Delhi.

```
cityMap.put("5", "New Delhi");
cityMap.put("5", "Kolkata");
```

36. Difference between ArrayList and LinkedList

In Java collections framework ArrayList and LinkedList are two different implementations of List interface (LinkedList also implement Deque interface though).

LinkedList is implemented using a doubly linked list concept where as ArrayList internally uses an array of Objects which can be resized dynamically. In most of the cases we do use ArrayList and it works very well but there are some use cases where use of the linked list may be a better choice. So let's see some of the difference between ArrayList and LinkedList, it will help you in making an informed choice when to use ArrayList and when a LinkedList.

ArrayList class provides a constructor ArrayList(int initialCapacity) even if the default constructor ArrayList() is used an empty list is constructed with an initial capacity of ten. Where as LinkedList just provides one constructor LinkedList() which constructs an empty list. Note that in LinkedList there is no provision of initial capacity or default capacity.

What it means is that if elements are added at the last every time and capacity is not breached then ArrayList will work faster because it already has an initial capacity and just need to add that element at the index in the underlying array. Where as in LinkedList a new node has to be created and references for Next and Prev are to be adjusted to accommodate the new Node.

The above mentioned case for add is something we do most of the time while using arraylist or use get(int index) which also is faster in arraylist and that's why we normally use ArrayList.

Let's go through some of the operations to see which implementation of list shines where –

36.1 Adding an element

If you are frequently adding element at the beginning of the list then LinkedList may be a better choice because in case of arraylist adding at the beginning every time will result in shuffling all the existing elements within the underlying array by one index to create space for the new element.

Even in the case of adding at the last ArrayList may give O(n) performance in the worst case. That will happen if you add more elements than the capacity of the underlying array, as in that case a new array (1.5 times the last size) is created, and the old array is copied to the new one.

So for LinkedList add(e Element) is always O(1) where as for ArrayList add(e Element) operation runs in amortized constant time, that is, adding n elements requires O(n) time.

36.2 Retrieving an element

Retrieving an element from the list using get(int index) is faster in ArrayList than in LinkedList. Since ArrayList internally uses an array to store elements so get(int index) means going to that index directly in the array. In Linked list get(int index) will mean traversing through the linked list nodes.

So for LinkedList get(int index) is O(n) where as for ArrayList get(int index) is O(1).

36.3 Removing an element

If you are removing using the remove(int index) method then for LinkedList class it will be O(n) as list has to be traversed to get to that index and then the element removed. Though the LinkedList provides methods like removeFirst() and removeLast() to remove the first or last element and in that case it will be O(1).

In case of ArrayList getting to that index is fast but removing will mean shuffling the remaining elements to fill the gap created by the removed element with in the underlying array. It ranges from O(1) for removing the last element to O(n). Thus it can be said remove(int index) operation is O(n - index) for the arraylist.

36.4 Removing an element while iterating

if you are iterating the list and removing element using iterator.remove() then for LinkedList it is O(1) as the iterator is already at the element that has to be removed so removing it means just adjusting the Next and Prev references. Where as for ArrayList it is again O(n - index) because of an overhead of shuffling the remaining elements to fill the gap created by the removed elements.

One more difference is that LinkedList implementation provides a descendingIterator() which comes from implementing the Deque interface. descendingIterator() returns an iterator over the elements in this deque in reverse sequence.

In ArrayList there is no such iterator.

37. Difference between ArrayList and Vector

Vector, just like ArrayList is also a growable dynamic array. As of Java v1.2, this class was retrofitted to implement the List interface, making it a member of the Java Collections Framework. With JDK 5 it was also retrofitted for generics and also implements Iterable interface which means it can also use enhanced for loop.

So you can see Vector in many ways is just like ArrayList apart from some differences and this post is about those difference between the ArrayList and Vector.

ArrayList Vs Vector in Java

1.ArrayList is not synchronized whereas Vector is Synchronized. Which means that the methods in the Vector class are Synchronized methods making Vector thread-safe and ready to use as-is in a multi-threaded environment.

ArrayList, if needed in a multi-threaded environment, has to be synchronized externally using Collections.synchronizedList method which returns a synchronized (thread-safe) list backed by the specified list.

2. A Vector, by default, doubles the size of its array when it needs to expand the array, while the ArrayList increases its array size by 50 percent.

3. As mentioned though Vector is retrofitted to implement the List interface it still has legacy methods which are not there in ArrayList. As Exp. addElement(), removeElement(),capacity() which are there in Vector class but not in ArrayList.

4. Performance wise Vector is comparatively slower than the ArrayList because it is synchronized. That means only one thread can access method of Vector at the time and there is also overhead of acquiring lock on the object.

5. For traversing an ArrayList as well as Vector, Iterator or ListIterator can be used. That Iterator/ListIterator is fail-fast and throws ConcurrentModificationException if any other Thread modifies the map structurally by adding or removing any element except Iterator's own remove() method.

For Vector enumeration can also be used for traversing which is not fail-fast.

38. TreeMap in Java

TreeMap is also one of the implementation of the Map interface like HashMap and LinkedHashMap. TreeMap class implements the NavigableMap interface and extends the AbstractMap class.

How it differs from other implementations of the Map interface like HashMap and LinkedHashMap is that objects in TreeMap are stored in sorted order. The elements are ordered using their natural ordering or a comparator can be provided at map creation time to provide custom ordering (We'll see an example a little later).

38.1 How TreeMap is implemented

TreeMap is a Red-Black tree based NavigableMap implementation. This implementation provides guaranteed log(n) time cost for the containsKey, get, put and remove operations.

Example code using TreeMap

```
public class TreeMapDemo {

    public static void main(String[] args) {
        Map<String, String> cityTemperatureMap = new TreeMap<String,
String>();
        // Storing elements
        cityTemperatureMap.put("Delhi", "24");
        cityTemperatureMap.put("Mumbai", "32");
        cityTemperatureMap.put("Chennai", "35");
        cityTemperatureMap.put("Bangalore", "22" );
        cityTemperatureMap.put("Kolkata", "28");
        cityTemperatureMap.put("Chennai", "36");

        // iterating the map
        for(Map.Entry<String, String> me :
cityTemperatureMap.entrySet()){
            System.out.println(me.getKey() + " " + me.getValue());
        }
    }
}
Output

Bangalore 22
Chennai 36
```

```
Delhi 24
Kolkata 28
Mumbai 32
```

It can be seen that the TreeMap sorts the elements using their natural ordering which is ascending for String.

Also note that though Chennai is added twice but it is stored only once as trying to store the same key twice will result in overwriting of the old value with the new value (as the calculated hash will be the same for the keys). Thus the last one is displayed while iterating the values.

38.2 TreeMap doesn't allow null

Though HashMap and LinkedHashMap allow one null as key, TreeMap doesn't allow null as key. Any attempt to add null in a TreeMap will result in a NullPointerException.

```
public class TreeMapDemo {

    public static void main(String[] args) {
        Map<String, String> cityTemperatureMap = new TreeMap<String,
String>();
        // Storing elements
        cityTemperatureMap.put("Delhi", "24");
        cityTemperatureMap.put("Mumbai", "32");
        cityTemperatureMap.put("Chennai", "35");
        cityTemperatureMap.put("Bangalore", "22" );
        cityTemperatureMap.put("Kolkata", "28");
        // Null key
        cityTemperatureMap.put(null, "36");

        // iterating the map
        for(Map.Entry<String, String> me :
cityTemperatureMap.entrySet()){
            System.out.println(me.getKey() + " " + me.getValue());
        }
    }
}
Output

Exception in thread "main" java.lang.NullPointerException
 at java.util.TreeMap.put(Unknown Source)
 at org.netjs.prog.TreeMapDemo.main(TreeMapDemo.java:17)
```

38.3 TreeMap is not synchronized

TreeMap is not thread safe. In case we need to Synchronize it, it should be synchronized externally. That can be done using the Collections.synchronizedSortedMap method.

```
SortedMap m = Collections.synchronizedSortedMap(new TreeMap(...));
```

38.4 TreeMap class' iterator is fail-fast

The iterators returned by this class' iterator method are fail-fast, if the map is structurally modified at any time after the iterator is created, in any way except through the iterator's own remove method, the iterator will throw a ConcurrentModificationException.

38.5 Sorting elements in different order in TreeMap

As already mentioned by default elements are stored in TreeMap using natural ordering. If you want to sort using different order then you need to provide your own comparator at map creation time. Let's see an example where we sort the Strings in descending order.

```
public class TreeMapDemo {

    public static void main(String[] args) {
        Map<String, String> cityTemperatureMap = new TreeMap<String,
String>(new TreeComparator());
        // Storing elements

        cityTemperatureMap.put("Delhi", "24");
        cityTemperatureMap.put("Mumbai", "32");
        cityTemperatureMap.put("Chennai", "35");
        cityTemperatureMap.put("Bangalore", "22" );
        cityTemperatureMap.put("Kolkata", "28");

        // iterating the map
        for(Map.Entry<String, String> me :
cityTemperatureMap.entrySet()){
            System.out.println(me.getKey() + " " + me.getValue());
        }

    }

}

//Comparator class
class TreeComparator implements Comparator<String>{
 @Override
 public int compare(String str1, String str2) {
     return str2.compareTo(str1);
 }
}
Output

Mumbai 32
Kolkata 28
Delhi 24
Chennai 35
Bangalore 22
```

Here note that a comparator is provided which reverses the sorting order. That comparator is provided at the map creation time in a constructor.