**Abstraction** is more about hiding the implementation details. In **Java abstraction** is achieved through **abstract** classes and interfaces.

A **Java** class, where all instance variables are private and only the methods with in the class can manipulate those variables, is an **example** of encapsulated class.

**The significant differences between extending Thread class and implementing Runnable interface:**

* When we extend Thread class, we can’t extend any other class even we require and When we implement Runnable, we can save a space for our class to extend any other class in future or now.
* When we extend Thread class, each of our thread creates unique object and associate with it. When we implements Runnable, it shares the same object to multiple threads.

**Hashtable** uses **single lock** for whole data.

**ConcurrentHashMap** uses **multiple locks** on segment level (16 by default) instead of object level i.e. whole Map.

* **Hashtable** locks home's main door. **ConcurrentHashMap** locks specific room door instead of main door.

**ConcurrentHashMap** can guarantee that there is no **ConcurrentModificationException** thrown while one thread is updating the map and another thread is traversing the iterator obtained from the map. However, **Collections.synchronizedMap**() is not guaranteed on this.

**Difference between HashMap and ConcurrentHashMap**

[**HashMap**](https://www.geeksforgeeks.org/java-util-hashmap-in-java/) is the Class which is under Collection and **ConcurrentHashMap** is a Class which is under Concurrent Collections, apart from this there are various differences between them which are:

* HashMap is non-Synchronized in nature i.e. HashMap is not Thread-safe whereas ConcurrentHashMap is Thread-safe in nature.
* HashMap performance is relatively high because it is non-synchronized in nature and any number of threads can perform simultaneously. But ConcurrentHashMap performance is low sometimes because sometimes Threads are required to wait on ConcurrentHashMap.
* While one thread is Iterating the HashMap object, if other thread try to add/modify the contents of Object then we will get Run-time exception saying **ConcurrentModificationException**. Whereas in ConcurrentHashMap we wont get any exception while performing any modification at the time of Iteration.

**Using HashMap**

import java.util.HashMap;

class HashMapDemo extends Thread

{

static HashMap<Integer,String> l=new HashMap<Integer,String>();

    public void run()

    {

        // Child thread trying to add

        // new element in the object

        l.put(103,"D");

        try

        {

            Thread.sleep(1000);

        }

        catch(InterruptedException e)

        {

            System.out.println("Child Thread going to add element");

        }

    }

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

    {

        l.put(100,"A");

        l.put(101,"B");

        l.put(102,"C");

        HashMapDemo t=new HashMapDemo();

        t.start();

        for (Object o : l.entrySet())

        {

            Object s=o;

            System.out.println(s);

            Thread.sleep(1000);

        }

        System.out.println(l);

    }

}

*chevron\_right*

*filter\_none*

Output:

100=A

Exception in thread "main" java.util.ConcurrentModificationException

**Using ConcurrentHashMap**

|  |
| --- |
| import java.util.HashMap;  import java.util.concurrent.\*;  class HashMapDemo extends Thread  {  static ConcurrentHashMap<Integer,String> l = new ConcurrentHashMap<Integer,String>();      public void run()      {          l.put(103,"D");    try          {              Thread.sleep(2000);          }          catch(InterruptedException e)          {              System.out.println("Child Thread going to add element");          }      }      public static void main(String[] args) throws InterruptedException      {          l.put(100,"A");          l.put(101,"B");          l.put(102,"C");          HashMapDemo t=new HashMapDemo();          t.start();          for (Object o : l.entrySet())          {              Object s=o;              System.out.println(s);              Thread.sleep(1000);          }          System.out.println(l);      }  } |

*chevron\_right*

*filter\_none*

Output:

100=A

101=B

102=C

103=D

{100=A, 101=B, 102=C, 103=D}

* In HashMap, null values are allowed for key and values, whereas in ConcurrentHashMap null value is not allowed for key and value, otherwise we will get Run-time exception saying **NullPointerException.**

# Static Block and main() method in Java

In Java [static block](https://www.geeksforgeeks.org/g-fact-79/) is used to initialize the static data members. Important point to note is that static block is executed before the main method at the time of class loading.

|  |
| --- |
| class staticExample {      // static block      static      {          System.out.println("Inside Static Block.");      }      // main method      public static void main(String args[])      {          System.out.println("Inside main method.");      }  } |

**Advantages of Rest Service Or Microservices**

High **scalability** – Demanding services can be deployed in multiple servers to enhance **performance** and keep away from other services so that they don't impact other services.

**Serialization Of Static**

serialVersionUID is a **special** static variable used by the serialization and deserialization process, to verify that a local class is compatible with the class used to serialize an object. It's not just a static variable as others, which are definitely not serialized.

When an object of a class is first serialized, a class descriptor containing among other things the class name and serial version UID is written to the stream. When this is deserialized, the JVM checks if the serial version UID read from the stream is the same as the one of the local class. If they're not, it doesn't even try to deserialize the object, because it knows the classes are incompatible.

private static final long serialVersionUID = 1L;

**If a static variable was not serialized then, we often face an exception during the de-serialization process.**

**java.io.InvalidClassException**

in which the serialVersionUID from the deserialized object is extracted and compared with the serialVersionUID of the loaded class.

An **abstract class** may contain non-**final variables**. Type of **variables**: **Abstract class can have final**, non-**final**, **static** and non-**static variables**. Interface **has** only **static and final variables**. ... A **Java abstract class can have class** members like private, protected, etc.