|  | **HashMap** | **ConcurrentHashMap** |
| --- | --- | --- |
| Synchronized | HashMap is not synchronized. | ConcurrentHashMap is synchronized. |
| Thread Safe | HashMap is not thread safe. | ConcurrentHashMap is thread safe. |
| Iterator type | HashMap iterator is fail-fast throws ConcurrentModificationException if concurrent modification happens during iteration. | ConcurrentHashMap is fail-safe and it will never throw ConcurrentModificationException during iteration. |
| Null values | HashMap allows key and value to be null. | ConcurrentHashMap does not allow null key/value. It will throw NullPointerException. |
| Performance | HashMap is faster. | ConcurrentHashMap is slower than HashMap. |
| Since Java Version | 1.2 | 1.5 |

### Why ConcurrentHashMap does not allow null keys and null values?

* ConcurrentHashMap does not allow null keys and null values
* The main reason that nulls aren't allowed in ConcurrentMaps because there will be ambiguities that may be just barely tolerable in non-concurrent maps can't be accommodated.
* **The main one is that if map.get(key) returns null, you can't detect whether the key explicitly maps to null vs the key isn't mapped.**
* In a non-concurrent map, you can check this via map.contains(key), but in a concurrent one, the map might have changed between calls.

The code is like this:

if (map.containsKey(k)) {

     return map.get(k);

} else {

     throw new KeyNotPresentException();

}

It might be possible that key k might be deleted in between the get(k) and containsKey(k) calls.

As a result, the code will return null as opposed to **KeyNotPresentException** (Expected Result if key is not present).

The Null key and value allowed in HashMap because **there is no Concurrent access**.

### Why can’t constructors be final, static, native, synchronized or abstract in Java?

### When you use a final keyword with a method or constructor it cannot be overridden. But, a constructor in Java cannot be overridden therefore; there is no need of using the final keyword with the constructor.

### Since you cannot override a constructor you cannot provide body to it if it is made abstract. Therefore, you cannot use abstract keyword with the constructor

### When you set a method as ‘static’, it means: “Method belong to class, not a particular object” but constructor implicitly called to initialize an object, so there is no purpose in having a static constructor.

### Can we override and overload static method

### No, you cannot override static method in Java, though you can declare method with same signature in sub class. It won't be overridden in exact sense, instead that is called method hiding.

### But at same time, you can overload static methods in Java, there is nothing wrong declaring static methods with same name, but different arguments

**Can we synchronized a static method**  🡪 yes. It will acquire the lock at class level.

### Methods in object class

1. **toString() :** toString() provides String representation of an Object and used to convert an object to String.

**2. hashCode() :** For every object, JVM generates a unique number which is hashcode. It returns distinct integers for distinct objects.

* Use of hashCode() method : Returns a hash value that is used to search object in a collection. JVM(Java Virtual Machine) uses hashcode method while saving objects into hashing related data structures like HashSet, HashMap, Hashtable etc. The main advantage of saving objects based on hash code is that searching becomes easy.
* Note : Override of hashCode() method needs to be done such that for every object we generate a unique number.
* hashCode() method of object class returns the memory reference of object in integer form. Definition of hashCode() method is public native hashCode(). It indicates the implementation of hashCode() is native because there is not any direct method in java to fetch the reference of object.

**3. equals(Object obj) :** Compares the given object to “this” object (the object on which the method is called)

* Note : It is generally necessary to override the hashCode() method whenever this method is overridden, so as to maintain the general contract for the hashCode method, which states that equal objects must have equal hash codes.

**4. getClass() :** Returns the class object of “this” object and used to get actual runtime class of the object.

* It can also be used to get metadata of this class. The returned Class object is the object that is locked by static synchronized methods of the represented class. As it is final so we don’t override it.

**5. finalize():** This method is called just before an object is garbage collected.

* It is called by the Garbage Collector on an object when garbage collector determines that there are no more references to the object. We should override finalize() method to dispose system resources, perform clean-up activities and minimize memory leaks. For example before destroying Servlet objects web container, always called finalize method to perform clean-up activities of the session.
* Note :finalize method is called just once on an object even though that object is eligible for garbage collection multiple times.

**6. clone() :** It returns a new object that is exactly the same as this object.

**7. wait():** causes the current thread to wait until another thread invokes the **notify()**or **notifyAll()**methods for that object.

**8. notify():** wakes up a single thread that is waiting on that object’s monitor

**9. notifyAll():**wakes up all threads that are waiting on that object’s monitor

A thread waits on an object’s monitor by calling one of the **wait()**method. These methods can throw **IllegalMonitorStateException** if the current thread is not the owner of the object’s monitor.

### Static and dynamic binding

1. **Static Binding**: The binding which can be resolved at compile time by compiler is known as static or early binding. Binding of all the static, private and final methods is done at compile-time .
2. **Why binding of static, final and private methods is always a static binding?**

Static binding is better performance wise (no extra overhead is required). Compiler knows that all such methods cannot be overridden and will always be accessed by object of local class. Hence compiler doesn’t have any difficulty to determine object of class (local class for sure). That’s the reason binding for such methods is static.

1. **Dynamic Binding**: In Dynamic binding compiler doesn’t decide the method to be called. Overriding is a perfect example of dynamic binding. In overriding both parent and child classes have same method
2. Private, final and static members (methods and variables) use static binding while for virtual methods (In Java methods are virtual by default) binding is done during run time based upon run time object.
3. Static binding uses Type information for binding while Dynamic binding uses Objects to resolve binding.
4. Overloaded methods are resolved (deciding which method to be called when there are multiple methods with same name) using static binding while overridden methods using dynamic binding, i.e, at run time.

### Transient and volatile

* Transient is a variables modifier used in serialization. **At the time of serialization, if we don’t want to save value of a particular variable in a file, then we use transient keyword. When JVM comes across transient keyword, it ignores original value of the variable and save default value of that variable data type.**
* Transient keyword plays an important role to meet security constraints. There are various real-life examples where we don’t want to save private data in file. Another use of transient keyword is not to serialize the variable whose value can be calculated/derived using other serialized objects or system such as age of a person, current date, etc.
* Practically we serialized only those fields which represent a state of instance, after all serialization is all about to save state of an object to a file. It is good habit to use transient keyword with private confidential fields of a class during serialization
* **Transient and static:** Since static fields are not part of state of the object, there is no use/impact of using transient keyword with static variables. However there is no compilation error.
* **Transient and final:** final variables are directly serialized by their values, so there is no use/impact of declaring final variable as transient. There is no compile-time error though.
* **Volatile** in Java is used as an indicator to Java compiler and Thread that do not cache value of this variable and always read it from main memory.
* The Java volatile keyword cannot be used with method or class and it can only be used with a variable.

### Hash collision

The phenomenon when two keys have same hash code is called hash collision. If hashCode() method is not implemented properly, there will be higher number of hash collision and map entries will not be properly distributed causing slowness in the get and put operations(as hashcode of key is used to decide the bucket and if hashcode is same for most then data will be more on one bucket). **This is the reason for prime number usage in generating hash code so that map entries are properly distributed across all the buckets**.

### How to redirect output of System.out to file.

We must reassign the standard output by using the following method of System class:

System.setOut(PrintStream p);

PrintStream can be used for character output to a text file

// Java program to demonstrate redirection in System.out.println()

import java.io.\*;

public class SystemFact

{

    public static void main(String arr[]) throws FileNotFoundException

    {

        // Creating a File object that represents the disk file.

        PrintStream o = new PrintStream(new File("A.txt"));

        // Store current System.out before assigning a new value

        PrintStream console = System.out;

        // Assign o to output stream

        System.setOut(o);

        System.out.println("This will be written to the text file");

        // Use stored value for output stream

        System.setOut(console);

        System.out.println("This will be written on the console!");

    }

}

### Association, composition, aggregation

* **Association** is relation between two separate classes which establishes through their Objects.
* **Aggregation** implies a relationship where the child can exist independently of the parent.

For example, Bank and Employee, delete the Bank and the Employee still exist.

* **Composition** implies a relationship where the child cannot exist independent of the parent.

Example: Human and heart, heart don’t exist separate to a Human

* **Type of Relationship**: Aggregation relation is “has-a” and composition is “part-of” relation.
* **Type of association:** Composition is a strong Association whereas Aggregation is a weak Association.

### Relation between hashcode and equals method

### 2 object if equals will have same hashcode but 2 objects with same hashcode may not be necessary that they are equal

### Top of exception hierarcy 🡪 Throwable

### Example of errors 🡪 stackoverflow or outofmemory

### How to sort external class in case we don’t have its implementation 🡪 comparator

### When to use comparable and comparator 🡪 if different sorting needed then comparator.

### Diff between throw and throws 🡪 throw in exception block and throws in method signature

### How to maintain an insertion order in map 🡪 linkedhashmap

1. Internal working of hashMap. Internal size and what is bucket
2. Internal working of ConcurrenthashMap. What is segment
3. When to use Abstract class n interface
4. java 8 features
5. why default methods in interface
6. what are wrapper classes
7. ClassNotFoundException Vs NoClassDefFoundError
8. diff between fail fast and fail safe
9. functional interface ? what all can we have in functional interface
10. lambda
11. HashSet work internally
12. how to handle hash collosion 🡪 implement hash code
13. locking concepts in threads
14. object level and class level lock
15. abstraction and encapsulation
16. super and this
17. Diff ways of creating object in java without new keyword
18. how do we handle runtime exception
19. doubly linked list
20. Statics in serialisation
21. Singeton and singeton scope diffrence
22. Generics and collection
23. Collection stream api
24. thread , runnable and callable
25. Comparable and Comparator with syntax
26. ConcurrentHashMap iteration
27. rehashing in java collection
28. Multithreading example in your project.
29. equals and hashcode method override, how to restrict subcalss from using super class equals method.
30. how java do overloading/overriding internally
31. how to create only 11 instacne of a class
32. How to handle ERROR in java.
33. can inner class extend any class or interface.
34. what will happen if variable is defined after inner class declaration.
35. double 10.0/0 and -10.0/0 ans
36. What if array is passed to another method and changed its value will it get reflected in original array.
37. What if class has defined public static void main in super class and sub class inherit it.
38. explain class loader and its types
39. how to read heap dump
40. anonymous inner class
41. jvm architecture
42. can we use super in interface
43. functional interface, static and default methods
44. encapsulation and abstraction
45. Mockito doThrow and doReturn
46. Generics in Java
47. List<Parent> p = new Arraylist<child> is this possible?
48. Data hiding concept with variable in java.
49. What modifiers can be used (abstract/final/public/static) with method local inner class.

## @SpringBootApplication

Spring boot is mostly about auto-configuration. This auto-configuration is done by **component scanning** i.e. finding all classes in classspath for [@Component](https://howtodoinjava.com/spring-core/how-to-use-spring-component-repository-service-and-controller-annotations/) annotation. It also involve scanning of @Configuration annotation and initialize some extra beans.

[@SpringBootApplication](https://howtodoinjava.com/spring-boot/springbootapplication-auto-configuration/) annotation enable all able things in one step. It enables the three features:

1. @EnableAutoConfiguration : enable auto-configuration mechanism
2. [@ComponentScan](https://howtodoinjava.com/spring-mvc/spring-mvc-difference-between-contextannotation-config-vs-contextcomponent-scan/) : enable @Component scan
3. @SpringBootConfiguration : register extra beans in the context

The java class annotated with @SpringBootApplication is the main class of a Spring Boot application and application starts from here.

|  |
| --- |
| import org.springframework.boot.SpringApplication;  import org.springframework.boot.autoconfigure.SpringBootApplication;    @SpringBootApplication  public class Application {       public static void main(String[] args) {          SpringApplication.run(Application.class, args);      }   } |

## @EnableAutoConfiguration

This annotation enables auto-configuration of the Spring Application Context, attempting to guess and configure beans that we are likely to need based on the presence of predefined classes in classpath.

For example, if we have tomcat-embedded.jar on the classpath, we are likely to want a TomcatServletWebServerFactory.

Auto-configuration classes are regular Spring Configuration beans. They are located using the SpringFactoriesLoader mechanism (keyed against this class). Generally auto-configuration beans are @Conditional beans (most often using @ConditionalOnClass and @ConditionalOnMissingBean annotations).

## @SpringBootConfiguration

It indicates that a class provides Spring Boot application configuration. It can be used as an alternative to the Spring’s standard @Configuration annotation so that configuration can be found automatically.

Application should only ever include one @SpringBootConfiguration and most idiomatic Spring Boot applications will inherit it from @SpringBootApplication.

The main difference is both annotations is that @SpringBootConfiguration allows configuration to be automatically located. This can be especially useful for unit or integration tests.

## 

## @ImportAutoConfiguration

It import and apply only the specified auto-configuration classes. The difference between @ImportAutoConfiguration and @EnableAutoConfiguration is that later attempts to configure beans that are found in the classpath during scanning, whereas @ImportAutoConfiguration only runs the configuration classes that we provide in the annotation.

We should use @ImportAutoConfiguration when we don’t want to enable the default auto-configuration.

|  |
| --- |
| **@ImportAutoConfiguration example** |
| @ComponentScan("path.to.your.controllers")  @ImportAutoConfiguration({WebMvcAutoConfiguration.class      ,DispatcherServletAutoConfiguration.class      ,EmbeddedServletContainerAutoConfiguration.class      ,ServerPropertiesAutoConfiguration.class      ,HttpMessageConvertersAutoConfiguration.class})  public class App  {      public static void main(String[] args)      {          SpringApplication.run(App.class, args);      }  } |

## @AutoConfigureBefore, @AutoConfigureAfter, @AutoConfigureOrder

We can use the @AutoConfigureAfter or @AutoConfigureBefore annotations if our configuration needs to be applied in a specific order (before of after).

If we want to order certain auto-configurations that should not have any direct knowledge of each other, we can also use @AutoConfigureOrder. That annotation has the same semantic as the regular @Order annotation but provides a dedicated order for auto-configuration classes.

|  |
| --- |
| @AutoConfigureAfter Example |
| @Configuration  @AutoConfigureAfter(CacheAutoConfiguration.class)  @ConditionalOnBean(CacheManager.class)  @ConditionalOnClass(CacheStatisticsProvider.class)  public class RedissonCacheStatisticsAutoConfiguration  {      @Bean      public RedissonCacheStatisticsProvider redissonCacheStatisticsProvider(){          return new RedissonCacheStatisticsProvider();      }  } |

## 

## Condition Annotations

All auto-configuration classes generally have one or more @Conditional annotations. It allow to register a bean only when the condition meets. Following are some useful conditional annotations to use.

#### 5.1. @ConditionalOnBean and @ConditionalOnMissingBean

These annotations let a bean be included based on the presence or absence of specific beans.

It’s value attribute is used to specify beans **by type** or by name. Also the search attribute lets us limit the ApplicationContext hierarchy that should be considered when searching for beans.

Using these annotations at the class level prevents registration of the @Configuration class as a bean if the condition does not match.

In below example, bean JpaTransactionManager will only be loaded if a bean of type JpaTransactionManager is not already defined in the application context.

|  |
| --- |
| @Bean  @ConditionalOnMissingBean(type = "JpaTransactionManager")  JpaTransactionManager transactionManager(EntityManagerFactory entityManagerFactory)  {      JpaTransactionManager transactionManager = new JpaTransactionManager();      transactionManager.setEntityManagerFactory(entityManagerFactory);      return transactionManager;  } |

#### 5.2. @ConditionalOnClass and @ConditionalOnMissingClass

These annotations let configuration classes be included based on the presence or absence of specific classes. Notice that annotation metadata is parsed by using spring ASM module, and even if a class might not be present in runtime – you can still refer to the class in annotation.

We can also use value attribute to refer the real class or the name attribute to specify the class name by using a String value.

Below configuration will create EmbeddedAcmeService only if this class is available in runtime and no other bean with same name is present in application context.

|  |
| --- |
| @Configuration  @ConditionalOnClass(EmbeddedAcmeService.class)  static class EmbeddedConfiguration  {        @Bean      @ConditionalOnMissingBean      public EmbeddedAcmeService embeddedAcmeService() { ... }    } |

#### 5.3. @ConditionalOnNotWebApplication and @ConditionalOnWebApplication

These annotations let configuration be included depending on whether the application is a “web application” or not. In Spring, a web application is one which meets at least one of below three requirements:

1. uses a Spring WebApplicationContext
2. defines a session scope
3. has a StandardServletEnvironment

#### 5.4. @ConditionalOnProperty

This annotation lets configuration be included based on the presence and value of a Spring Environment property.

For example, if we have different datasource definitions for different environments, we can use this annotation.

|  |
| --- |
| @Bean  @ConditionalOnProperty(name = "env", havingValue = "local")  DataSource dataSource()  {      // ...  }    @Bean  @ConditionalOnProperty(name = "env", havingValue = "prod")  DataSource dataSource()  {      // ...  } |

#### 5.5. @ConditionalOnResource

This annotation lets configuration be included only when a specific resource is present in the classpath. Resources can be specified by using the usual Spring conventions.

|  |
| --- |
| @ConditionalOnResource(resources = "classpath:vendor.properties")  Properties additionalProperties()  {      // ...  } |

#### 5.6. @ConditionalOnExpression

This annotation lets configuration be included based on the result of a [SpEL expression](https://docs.spring.io/spring/docs/5.1.8.RELEASE/spring-framework-reference/core.html" \l "expressions). Use this annotation when condition to evaluate is complex one and shall be evaluated as one condition.

|  |
| --- |
| @Bean  @ConditionalOnExpression("${env} && ${havingValue == 'local'}")  DataSource dataSource()  {      // ...  } |

#### 5.7. @ConditionalOnCloudPlatform

This annotation lets configuration be included when the specified cloud platform is active.

|  |
| --- |
| @Configuration  @ConditionalOnCloudPlatform(CloudPlatform.CLOUD\_FOUNDRY)  public class CloudConfigurationExample  {    @Bean    public MyBean myBean(MyProperties properties)    {      return new MyBean(properties.getParam);    }  } |