What causes OutOfMemory exception in Java?

If there is no memory space for creating a new object in Heap, Java Virtual Machine throws OutOfMemoryError or more specifically**java.lang.OutOfMemoryError**heap space.****

**OutOfMemoryError exception**. Usually, this **error** is thrown when there is insufficient space to allocate an object in the **Java** heap. In this case, The garbage collector cannot make space available to accommodate a new object, and the heap cannot be expanded further.

How do I debug a memory error?

Steps to **debug** OutOfMemory **Error**  
  
Enable auto heap dump with the help of configuration on application startup. When an **Error** occurs, JVM will automatically create a heap dump file. You can also take a heap dump at any time when your application running(optional). Analyze the heap dump file using Eclipse **Memory** Analyzer.

Can memory leak in Java?

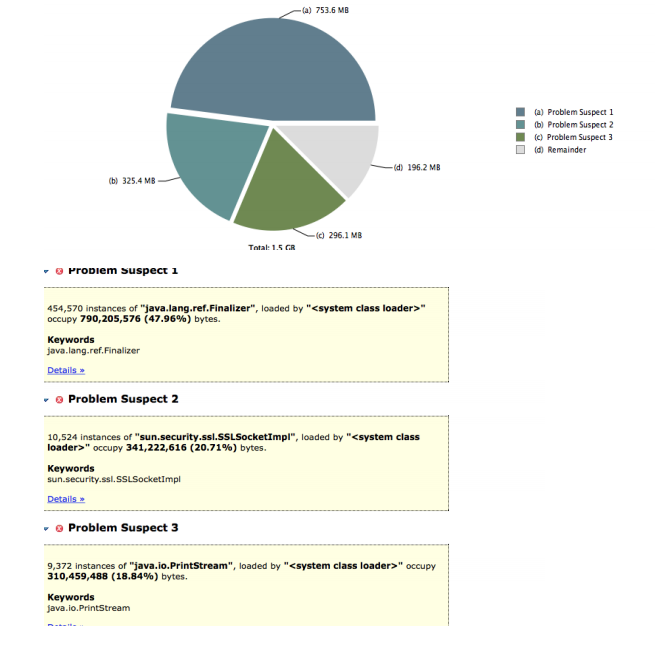
**Java** does automatic Garbage collection. ... Just because every objects has valid references, garbage collector in **Java can**'t destroys the objects. Such types of useless objects are called as **Memory leaks**. If allocated **memory** goes beyond limit, program will be terminated by rising OutOfMemoryError.

What is a **Memory Leak in Java**? The standard definition of a **memory leak** is a scenario that occurs when objects are no longer being used by the application, but the Garbage Collector is unable to remove them from working **memory** – because they're still being referenced.

How do you analyze out of memory error in Java?

**Heap Analysis Using MAT**

1. Open the heap (.hprof) generated when the **OOM error** happened. ...
2. This opens the dump with options for Leak Suspect Reports and Component Reports. ...
3. When the leak suspect chart opens, the pie in the overview pane shows the distribution of retained **memory** on a per-object basis.



### **Step 1: Increase Available Memory**

1. Load Only as Much Data as You Need.

How do you release memory in Java?

**Java** uses managed **memory**, so the only way you can allocate **memory** is by using the new operator, and the only way you can deallocate **memory** is by relying on the garbage collector

# Difference between Stack and Heap memory in Java

JVM has divided memory space between two parts one is Stack and another one is Heap space. Stack space is mainly used for storing order of method execution and local variables.

Stack always stored blocks in LIFO order whereas heap memory used dynamic allocation for allocating and deallocating memory blocks.

Memory allocated to the heap lives until one of the following events occurs :

* Program terminated
* Memory free

|  |  |
| --- | --- |
| Stack | Heap |
| Basic : Stack memory is used to store items which have a very short life like local variables, a reference variable of objects | Heap memory is allocated to store objects and JRE classes. |
| Ordering :The stack is always reserved in a LIFO (last in first out) order | Heap memory is dynamic allocation there is no fixed pattern for allocating and deallocating blocks in memory |
| Size :We can increase stack memory size by using JVM parameter -XSS | We can increase or decrease heap memory size by using JVM option -Xms and -Xmx |
| Visibility :Variables are visible to only to owner thread | It is visible to all threads |
| Exception :JVM will throw java.lang.StackOverFlowError | JVM will throw java.lang.OutOfMemoryError |

## **Heap Overflow**

Heap is used to store dynamic variables. It is a region of process’s memory. malloc(), calloc(), resize() all these inbuilt functions are generally used to store dynamic variables.

Heap overflow occurs when −

A) If we allocate dynamic large number of variables −

int main() {

   float \*ptr = (int \*)malloc(sizeof(float)\*1000000.0));

}

B) If we continuously allocate memory and do not free after using it.

int main() {

   for (int i=0; i<100000000000; i++) {

      int \*p = (int \*)malloc(sizeof(int));

   }

}

## **Stack Overflow**

The stack is a Last in First out data structure. It is used to store local variables which is used inside the function. Parameters are passed through this function and their return addresses.

If a program consumes more memory space, then stack overflow will occur as stack size is limited in computer memory.

Stack overflow occurs when

A) If a function is called recursively by itself infinite times then stack will be unable to store large number of local variables, so stack overflow will occur −

void calculate(int a) {

   if (a== 0)

      return;

      a = 6;

      calculate(a);

}

int main() {

   int a = 5;

   calculate(a);

}

B) If we declare a large number of local variables or declare a large dimensional array or matrix can result in stack overflow.

int main() {

   A[20000][20000];

}

## **Overflow**

Overflow occurs when we assign such a value to a variable which is more than the maximum permissible value.

## **Underflow**

Underflow occurs when we assign such a value to a variable which is less than the minimum permissible value.

JVM does not throw any exception in case Overflow or underflow occurs, it simply changes the value. Its programmer responsibility to check the possibility of an overflow/underflow condition and act accordingly.

****StackOverflowError Vs OutOfMemoryError in Java****

****StackOverflowError in Java****

****A JVM stack is created for each JVM thread. Whenever a method is invoked a new frame is created and pushed into the JVM stack for the thread. Each frame stores data related to the method like local variables, operand stack and a reference to the run-time constant pool of the class of the current method. Once the method execution is completed stack frame (for that method) is popped out of the stack.****

****If execution of any method requires a larger stack than is permitted, the Java Virtual Machine throws a StackOverflowError. You may see StackOverflowError when you have a recursive method with no terminating condition.****

For example a Java program to recursively print even numbers without any terminating condition.

public class StatckOverFlowErrorExample {

public static void main(String[] args) {

printEven(1);

}

private static int printEven(int i) {

if(i % 2 != 0) {

i++;

}

System.out.println(i);

return i + printEven(i + 2);

}}

OutOfMemoryError in Java

Whenever you create a new object memory for that object is allocated on the heap. Instance variables and arrays are also stored on the heap.

Once the object stored on the heap is not having any reference, memory for that object is reclaimed by garbage collector. If there are references to the object then GC can’t remove those objects, if you have large number of such referenced objects and JVM tries to allocate heap memory for new object, JVM throws java.lang.OutOfMemoryError because there is no sufficient heap memory left.

Trying to allocate an array that is larger than the heap size also results in OutOfMemoryError.

public static void main(String[] args) {

Integer[] array = new Integer[1000\*1000\*1000];}

|  |  |
| --- | --- |
| StackOverflowError | OutOfMemoryError |
| StackOverflowError is thrown when the *stack is full*. | OutOfMemoryError is thrown when the *heap space is full*. |
| Stack is used to store method related data when any method is invoked. So, StackOverflowError is thrown when there is no space left for storing method data. | Heap memory is used to store objects, instance variables and arrays. So, OutOfMemoryError is thrown when no space left for creating new objects, arrays. |
| Recursive methods with out terminating condition causes StackOverflowError. | Lots of objects with live references so GC can’t deallocate those objects results in OutOfMemoryError. |
| To avoid StackOverflowError ensure that methods are executing as per logic and terminating so that the stack frames for the executed methods can be taken out of the stack. | To avoid OutOfMemoryError ensure objects which are no longer required are not reference from any where and can be garbage collected. Also ensure not to create very large objects or arrays. |
|  |  |

### **Heap and OutOfMemoryError**

****OutOfMemoryError**** is related to ****Heap****.

If you have large objects (or) referenced objects in memeory, then you will see ****OutofMemoryError****. If you have strong references to objects, then GC can’t clean the memory space allocated for that object. When JVM tries to allocate memory for new object and not enough space available it throws ****OutofMemoryError**** because it can’t allocate required amount of memory.

### **Stack and StackOverflowError**

****StackOverflowError**** is related to ****stack****.

All your local variables and methods calls related data will be on stack. For every method call, one stack frame will be created and local as well as method call related data will be placed inside the stack frame. Once method execution is completed, stack frame will be removed. ONE WAY to reproduce this is, have infinite loop for method call, you will see stackoverflow error, because stack frame will be populated with method data for every call but it won’t be freed (removed).

## **Java Stack vs Heap**

### **Java Heap Space**

Java Heap space is used by java runtime to allocate memory to ****Objects and JRE classes****.

Whenever we create an object, it’s always created in the Heap space. ****Garbage Collection**** runs on the heap memory to free the memory used by objects that doesn’t have any reference. Any object created in the heap space has global access and can be referenced from anywhere of the application.

### **Java Stack Memory**

Java Stack memory is used for execution of a thread. They contain ****method specific values**** that are short-lived and ****references**** to other objects in the heap that are getting referred from the method.

Stack memory is always referenced in LIFO (Last-In-First-Out) order. Whenever a method is invoked, a new block is created in the stack memory for the method to hold local primitive values and reference to other objects in the method. As soon as method ends, the block becomes unused and become available for next method.  
Stack memory size is very less compared to Heap memory.

# Why String is Immutable in Java

Java String class is immutable which means once a String object is created it cannot be changed. When you use a String modification method like concatenation what actually happens is that a new String is created and returned that contains the result of the operation.

For example suppose there is String str-

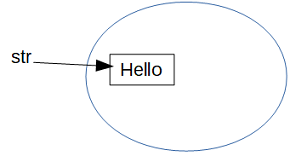
#### **Java String objects are immutable**

Java String class is immutable which means once a String object is created it cannot be changed. When you use a String modification method like concatenation what actually happens is that a new String is created and returned that contains the result of the operation.

For example suppose there is String str-

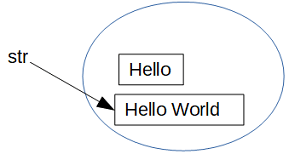
String str = “Hello”;

That means str refers to the memory location where value “Hello” is stored.



Now if you concatenate another value to this String and assign it to the same reference. Since original string can’t be modified because of String being immutable so this concatenation means a new String object is created with the modified value and str starts pointing to this new object.

str = str.concat(" World");



As you can see str now refers to the modified object and old object is unreferenced and ready to be garbage collected.

Here is another example where concatenation is done but the returned modified string is not assigned to any variable.

public class StringLiteral {

public static void main(String[] args) {

String str = "Hello";

str.concat(" World");

System.out.println("Value- " + str);

}}

****Output****

Value- Hello

As you can see str is still referring to the original String object.

#### **Why is String immutable in Java**

Now when we have seen with examples, what does String is immutable actually means let’s go through the why part.

1- ****Reduce memory usage****– To understand how does String being immutable results in reduced memory usage you’ll have to understand [String pool in Java](https://knpcode.com/java/java-basics/constant-string-pool-java/" \t "https://knpcode.com/java/java-basics/why-string-immutable-java/_blank).

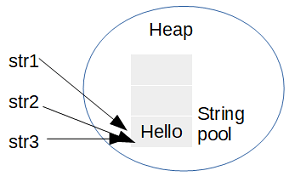
String is a special class in Java and also one of the most used too. That is why the concept of constant String pool is used to minimize memory usage.

Whenever any String literal is created (String enclosed in double quotes) JVM scans the String pool for any String having the same value, if found same reference is returned. So, new memory is not allocated for the same string, existing memory is used instead.

Let’s understand with an example. if three Strings are created as follows-

String str1 = “Hello”;String str2 = “Hello”;String str3 = “Hello”;

Then the same memory is referenced by all the three objects.



Now suppose the value of str1 is changed, if the original string itself is changed then what about the other two references. They will also start pointing to the changed value which is not correct. That is why making String immutable ensures that original object can’t be modified.  
Pooling strings is possible because of String in Java being immutable and that is how this property contributes in reduced memory usage by Strings.

2- ****Thread safety****– String in Java is immutable so that it can’t be modified once created which in turn means Strings are safe to be used in a multi-threaded environment with out any fear of change.  
If a thread changes the value of the shared string even then a new String is created leaving the original as unchanged.

3- ****Hashcode caching****– Another reason that can be given for why String is immutable in Java is that it enables hashcode to be cached for Strings.  
Since string once created can’t be modified so the hashcode once calculated for any string can be cached and reused. It is not required to recalculate that hashcode. That makes it very efficient to use String as a key in a hash based data structure like [HashMap](https://knpcode.com/java/collections/hashmap-in-java/" \t "https://knpcode.com/java/java-basics/why-string-immutable-java/_blank).

# Constant String Pool in Java

When a String object is created using string literal (value enclosed in double quotes) that String object is created in a part of memory known as ****constant string pool in Java****. In this tutorial you’ll learn what is string pool in Java and how it optimizes memory when Strings are created.

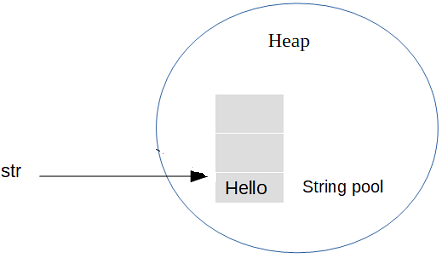
#### **Java constant String pool**

There are two ways to create a [String in Java](https://knpcode.com/java/java-basics/java-string-class-with-method-examples/" \t "https://knpcode.com/java/java-basics/constant-string-pool-java/_blank)–

* Using String literal
* Using new keyword

When you create a String using String literal, for example

String str = “hello”

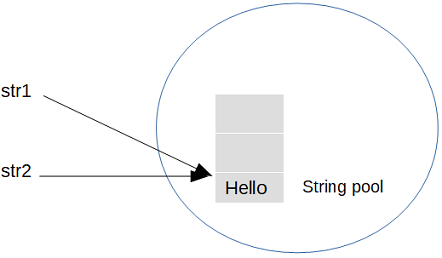
Memory for this string is created in a memory area knows as string pool in Java. This String pool is part of the heap memory.  
  
Now the question is why this String pool is required and how it optimizes memory usage? String is a special class in Java and one of the most used too. That is why the concept of constant String pool is used to minimize memory usage.

Whenever any String literal is created JVM scans the String pool for any String having the same value, if found same reference is returned. So, new memory is not allocated for the same string, existing memory is used instead.

Let’s understand with an example. if two Strings are created as follows-

String str1 = “Hello”;String str2 = “Hello”;

Then the same memory is referenced by both the objects.

  
You can also verify it using a Java program. In the example two Strings are created using string literals and then their references are compared using [equality ‘==’ operator](https://knpcode.com/java/java-basics/difference-between-equality-operator-and-equals-method-java/" \t "https://knpcode.com/java/java-basics/constant-string-pool-java/_blank).

public class StringLiteral {

public static void main(String[] args) {

String str1 = "Hello";

String str2 = "Hello";

// checking if memory reference is same

if(str1 == str2){

System.out.println("str1 and str2 are pointing to same memory reference");

}else{

System.out.println("str1 and str2 are not pointing to same memory reference");

}

}}

****Output****

str1 and str2 are pointing to same memory reference

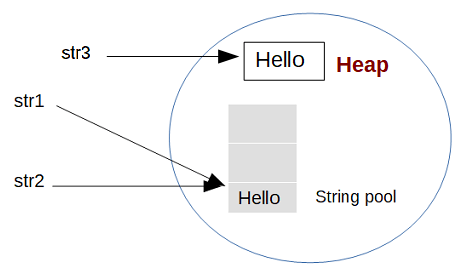
This efficient usage of memory through String pool is possible only because [String is immutable in Java](https://knpcode.com/java/java-basics/why-string-immutable-java/" \t "https://knpcode.com/java/java-basics/constant-string-pool-java/_blank).

#### **Creating String using new operator**

When a String instance is created using new operator, memory for the new instance is created on the heap rather than in the String pool.

For example if three Strings are created as follows-

String str1 = “Hello”;String str2 = “Hello”;String str3 = new String(“Hello”);

str1 and str2 share the same reference in the constant string pool where as str3 references to a memory location on the heap.  
  
You can also verify it using a Java program. In the example three Strings are created, two using string literals and one using new operator. Their references are compared using equality ‘==’ operator.

public class StringLiteral {

public static void main(String[] args) {

String str1 = new String("Hello");

String str2 = "Hello";

String str3 = "Hello";

// instance and literal

if(str1 == str2) {

System.out.println("str1 and str2 are pointing to same memory reference");

}else{

System.out.println("str1 and str2 are not pointing to same memory reference");

}

// instance and literal

if(str1 == str3) {

System.out.println("str1 and str3 are pointing to same memory reference");

}else{

System.out.println("str1 and str3 are not pointing to same memory reference");

}

// literal and literal

if(str2 == str3) {

System.out.println("str2 and str3 are pointing to same memory reference");

}else{

System.out.println("str2 and str3 are not pointing to same memory reference");

}

}}

****Output****

str1 and str2 are not pointing to same memory reference

str1 and str3 are not pointing to same memory reference

str2 and str3 are pointing to same memory reference

If two instances having same value are created using new operator even then these two objects will be allocated different memory.

public class StringLiteral {

public static void main(String[] args) {

String str1 = new String("Hello");

String str2 = new String("Hello");

if(str1 == str2) {

System.out.println("str1 and str2 are pointing to same memory reference");

}else{

System.out.println("str1 and str2 are not pointing to same memory reference");

}

}}

****Output****

str1 and str2 are not pointing to same memory reference

#### **Java String interning**

This process of sharing the memory is called interning in Java. In Java, string literals are “interned” so as to share unique instances, using the String.intern method. For String literals this process is implicit.

#### **Explicitly using intern method**

You can also intern a String by explicitly using intern method. That way you can even use the String pool for Strings created using new operator.

When the intern method is invoked, if the pool already contains a string equal to this String object as determined by the equals(Object) method, then the string from the pool is returned. Otherwise, this String object is added to the pool and a reference to this String object is returned.

public class StringLiteral {

public static void main(String[] args) {

String str1 = "Hello";

String str2 = new String("Hello");

// created using intern

String str3 = new String("Hello").intern();

// str3 = str2.intern(); can be created this way too

if(str1 == str2) {

System.out.println("str1 and str2 are pointing to same memory reference");

}else{

System.out.println("str1 and str2 are not pointing to same memory reference");

}

if(str1 == str3) {

System.out.println("str1 and str3 are pointing to same memory reference");

}else{

System.out.println("str1 and str3 are not pointing to same memory reference");

}

if(str2 == str3) {

System.out.println("str2 and str3 are pointing to same memory reference");

}else{

System.out.println("str2 and str3 are not pointing to same memory reference");

}

}}

****Output****

str1 and str2 are not pointing to same memory reference

str1 and str3 are pointing to same memory reference

str2 and str3 are not pointing to same memory reference

As you can see String str3 is created using intern method so it will use the String pool. Since there is already a String with same value in the pool so str3 uses the same reference that is why str1 and str3 have the same reference.

# Covariant Return Type in Java

[Method overriding in Java](https://knpcode.com/java/java-basics/method-overriding-in-java/" \t "https://knpcode.com/java/covariant-return-type-in-java/_blank) is said to be ****covariant**** with respect to return type which means that the return type is allowed to vary in the same direction as the subclass. Covariant return type feature in Java was added in Java 5.

#### **What is covariant return type in Java**

First let’s try to understand what exactly does this covariant return type mean? Before Java 5 it was not possible to change the return type of the overridden method in the sub-class meaning the method in the parent class and the overridden method in the child class should have the same name, same number and type of arguments and same return type.  
Java 5 onward, because of this covariant return type feature, it is possible for the overridden method to have different return type from the method in the parent class. There is one restriction though, *the return type of the sub-class method must be a subtype of the return type of the parent class method*.

****For example****– In parent class there is a method with return type R1. In the sub-class overridden method may have a different return type R2 but R2 must be a subtype of R1.

#### **Java covariant return type example**

class Parent{

Parent getInstance(){

return new Parent();

}

void parentTest(){

System.out.println("In parent class method");

}}class Child extends Parent{

// return type is subtype here

Child getInstance(){

return new Child();

}

void childTest(){

System.out.println("In child class method");

}}public class CovariantDemo {

public static void main(String[] args) {

Parent pobj = new Parent();

Child cobj = new Child();

pobj.getInstance().parentTest();

cobj.getInstance().childTest();

}}

****Output****

In parent class methodIn child class method

# HashSet Internal Implementation in Java

HashSet internal implementation in Java or how does HashSet work internally in Java is a very important interview question. Some of the important points that you should know are-

1. What is the backing data structure for HashSet or where does HashSet stores its element?
2. How does add() method work in HashSet?
3. How does remove() method work in HashSet?
4. How elements are retrieved from HashSet?

In this post we’ll go through the internal implementation of [HashSet in Java](https://knpcode.com/java/collections/hashset-in-java/" \t "https://knpcode.com/java/collections/hashset-internal-implementation-in-java/_blank) and try to explain the above mentioned points. Note that all the code snippet of the HashSet class provided in this post are from JDK 10.

Since HashSet internally uses HashMap for its operations so knowing [HashMap Internal Implementation in Java](https://knpcode.com/java/collections/hashmap-internal-implementation-in-java/" \t "https://knpcode.com/java/collections/hashset-internal-implementation-in-java/_blank) will help a lot in understanding internal implementation of HashSet.

#### **Where does HashSet store its element**

Internally HashSet in Java uses [HashMap](https://knpcode.com/java/collections/hashmap-in-java/" \t "https://knpcode.com/java/collections/hashset-internal-implementation-in-java/_blank) to store its elements. With in HashSet class a HashMap is defined that is used to store its elements.

private transient HashMap<E,Object> map;

If you see all the defined constructors for HashSet, all of those constructors create a HashMap.

public HashSet() {

map = new HashMap<>();}

public HashSet(Collection<? extends E> c) {

map = new HashMap<>(Math.max((int) (c.size()/.75f) + 1, 16));

addAll(c);}

public HashSet(int initialCapacity, float loadFactor) {

map = new HashMap<>(initialCapacity, loadFactor);}

public HashSet(int initialCapacity) {

map = new HashMap<>(initialCapacity);}

#### **Initial capacity, load factor and buckets in HashSet**

You should have clear understanding of the terms initial capacity, load factor and buckets to understand internal implementation of HashSet better.

As already mentioned HashSet uses HashMap to store its elements and HashMap in turn internally uses an array of type Node to store elements where Node<K, V> is an inner class with in HashMap class.

* ****Capacity****– If you don’t specify any capacity while creating HashSet then the array will have default initial capacity of 16. If you use the constructor where initial capacity is also passed then the array will have the specified initial capacity.
* ****Bucket****– In HashMap concept of bucket is used for storing elements where each index of array is conceptualized as one bucket. So, total there are 16 (in default case) buckets. For every (value) that is added to HashSet a hash is calculated using the key, based on that hash value one of these buckets is chosen to store the element.
* ****Load factor****– Load factor is the threshold for the HashSet storage. Once the threshold is reached the capacity of the HashSet is doubled. Default load factor is 0.75 which means if the 75% of the capacity is reached the HashSet is resized.

#### **How does add method work in Java HashSet**

You must be thinking if internally HashSet uses HashMap for adding elements then how come ****add(E e)**** method in HashSet takes only value as argument not a (key, value) pair. After all HashMap stores its element as (key, value) pair.

In Java HashSet implementation; from the ****add(E e)**** method, ****put()**** method of HashMap is called for adding the element and a (key, value) pair is sent too from HashSet. What internally happens is that the value passed for adding to the HashSet becomes the key for HashMap and a dummy object “PRESENT” is always added as value.

Dummy object PRESENT is defined in HashSet implementation as follows-

// Dummy value to associate with an Object in the backing Mapprivate static final Object PRESENT = new Object();

The implementation of add(E e) method is as follows-

public boolean add(E e) {

return map.put(e, PRESENT)==null;}

Here you can see that value passed for storing in HashSet becomes the key in the HashMap. Actually that’s how it’s ensured that only unique values are stored in HashSet. In HashMap value may be duplicate but Key should be unique. As we have seen that the value becomes key in HashMap which remains unique.

#### **How values are retrieved from HashSet**

There is no method in HashSet to get an individual value. You can [iterate over the HashSet](https://knpcode.com/java/collections/different-ways-to-iterate-a-hashset-in-java/" \t "https://knpcode.com/java/collections/hashset-internal-implementation-in-java/_blank) and get all the values though. The iterator method of the HashSet returns the keySet of the backing HashMap. We have already seen the values added to the HashSet becomes key in the HashMap so what you actually get is the values of the HashSet.

****keySet()****– Returns a Set view of the keys contained in this map.

The implementation of ****iterator()**** method is as follows-

public Iterator<E> iterator() {

return map.keySet().iterator();}

#### **How values are removed from HashSet**

For removing the value same interchange happens. What you provide as value for removing in the ****remove()**** method of the HashSet becomes the key while making a call to backing HashMap’s remove() method.

public boolean remove(Object o) {

return map.remove(o)==PRESENT;}

Here note that the remove method of the HashMap returns the value associated with key. Now we know that the value is always passed as “PRESENT” while adding to HashMap, that’s why the comparison ****map.remove(o)==PRESENT;****

#### **Important points**

1. HashSet is backed by a HashMap instance.
2. In the internal implementation of the HashSet a dummy object “PRESENT” is always added a value to the backing HashMap. The value passed to add to HashSet becomes key in the HashMap.
3. When the hash is calculated for HashSet it is calculated using the value itself as value has become in the HashMap.