IdentityHashMap is another type of Map that implements Map, Serializable, and Cloneable interfaces and extends the AbstractMap class. The main feature of this map is that while storing elements, the equality of the keys is checked on the basis of reference instead of the equals method. What this means is that if we have two keys, **key1** and **key2**, then **key1** will be considered equal to **key2** if key1 == key2. In other words, if both the keys reference the same object.

This means that **IdentityHashMap** intentionally violates Map's general contract which mandates the use of the equals method when comparing objects. This class is designed for use only in rare cases wherein reference-equality semantics are required.

Some of the features of **IdentityHashMap** are:

1. The IdentityHashMap stores the elements in random order.
2. The **IdentityHashMap** allows a single null key.
3. The **IdentityHashMap** is not thread-safe.

Difference between HashMap and IdentityHashMap

Let's discuss some of the differences between a HashMap and IdentityHashMap.

1. IdentityHashMap uses reference equality to compare keys and values while HashMap uses object equality to compare keys and values.
2. IdentityHashMap does not use the `hashCode()` method. Instead it uses `System.identityHashCode()` to find the bucket location.
3. IdentityHashMap does not require keys to be immutable as it does not rely on the `equals()` and `hashCode()` methods. To safely store the object in HashMap, keys must be immutable.
4. The default initial capacity of HashMap is 16; whereas, for IdentityHashMap, it is 32.

These are the major differences between a **HashMap** and **IdentityHashMap**. All the operations such as create, add, fetch, remove, etc in an **IdentityHashMap** are the same as these operations in hashMap, so we won't be discussing it here again. We will be looking at an example to understand the difference in how HashMap and **IdentityHashMap** run.

First, let's create an `Employee` class that overrides both the `equals()` and the `hashCode()` method.

```java
public class Employee {

    int empId;
    String empName;

    public Employee(int empId, String empName) {
        super();
        this.empId = empId;
        this.empName = empName;
    }

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

    @Override
    public boolean equals(Object obj) {
        Employee other = (Employee) obj;
        if (empId != other.empId)
            return false;
        if (empName == null) {
            if (other.empName != null)
                return false;
```

Now we will create two Employee objects with exactly the same data and will store them in an **IdentityHashMap** as well as in a HashMap. We will find out that an **IdentityHashMap** will store both the objects whereas the HashMap will have only one object, stored in it.

IdentityHashMapDemo.java

Employee.java

```java
import java.util.HashMap;
import java.util.IdentityHashMap;
import java.util.Map;

public class IdentityHashMapDemo {

    public static void main(String args[]) {

        Employee emp1 = new Employee(123, "Saurav");
        Employee emp2 = new Employee(123, "Saurav");


        Map<Employee, String> hashMap = new HashMap<>();
        hashMap.put(emp1, "emp1");
        hashMap.put(emp2, "emp2");


        Map<Employee, String> identityHashMap = new IdentityHashMap<>();
        identityHashMap.put(emp1, "emp1");
        identityHashMap.put(emp2, "emp2");

        System.out.println("The employee objects in HashMap are:");
        System.out.println(hashMap);

        System.out.println();
        System.out.println("The employee objects in IdentityHashMap are:");
        System.out.println(identityHashMap);
    }
```

Run Save Reset