The first and foremost requirement for a good key is that it should follow the `hashcode()` and `equals()` contract. The contract says:

1. If two objects are equal, then they must have the same hash code.
2. If two objects have the same hash code, they may or may not be equal.

This means that the class that is being used as a key must override both `equals()` and `hashcode()` methods.

Why overriding both `hashcode()` and `equals()` is important

If a class does not override both the `hashcode()` and `equals()` method, then it will break the contract and the **HashMap** may not work. Let's look at an example. We have an Employee class that has two fields as shown below:

```
public class Employee {
    int empId;
    String empName;
}
```

This class overrides the `hashcode()` method but does not override the `equals()` method. Ideally, two objects are considered equal if their `empId` is equal.

Now we will create two `Employee` objects with the same `empId` and `empName`. We will also create a **HashMap** where the key will be the Employee object, and the value will be the salary. The **HashMap** should not allow both the `Employee` objects to be inserted as they are equal.

HashMapDemo.java

Employee.java

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

public class HashMapDemo {

    public static void main(String args[]) {

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

        Map<Employee, Integer> employeeMap = new HashMap<>();

        employeeMap.put(emp1, 56000);
        employeeMap.put(emp2, 45000);

        for(Entry<Employee, Integer> entry : employeeMap.entrySet()) {
            System.out.println("Employee Id: " + entry.getKey().empId + " Emp
        }

    }
}
```

Run Save Reset

On running the above program, we can see that both the `Employee` objects got inserted in the **HashMap**. The reason is that since we have not overridden the `equals()` method, the `equals()` method of the Object class is called because Object is the superclass of all the classes. Below is the implementation of the `equals()` method in the Object class.

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

As we can see, it compares two reference points to the same object. Since we have created two separate `Employee` objects, the equality check failed and both the objects were saved.

Now, we will override the `equals()` method as well in the Employee class, and then we will see that only one Employee object is stored.

Employee.java

HashMapDemo.java

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

public class HashMapDemo {

    public static void main(String args[]) {

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

        Map<Employee, Integer> employeeMap = new HashMap<>();

        employeeMap.put(emp1, 56000);
        employeeMap.put(emp2, 45000);

        for(Entry<Employee, Integer> entry : employeeMap.entrySet()) {
            System.out.println("Employee Id: " + entry.getKey().empId + " Emp
        }

    }
}
```

Run Save Reset

Why immutable objects make a good key

It is not mandatory for a **HashMap** key to be immutable, but it is suggested that key objects are immutable. Immutability allows us to get the same hash code every time for a key object.

All the wrapper classes such as String, Integer, etc., are immutable, so they are considered good key candidates.

Let's look at an example to understand what happens when a key object is not immutable. In the below example, we are using an `Employee` object as a key. After inserting the employee object into the **HashMap**, we will make a change in the `Employee` object. After that, we will try to get the value for this key from the **HashMap**.

HashMapDemo.java

Employee.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 emp = (Employee) obj;
        return this.empId == emp.empId;
    }

}
```

Run Save Reset

When we run the above program, null is returned. The reason is when we change the Employee object its hashcode also changes. Therefore, when we try to search for the Employee object, a different bucket is returned. So, although the object was present in the **HashMap**, it is not returned.

If we are using a custom object as the **HashMap** key, then either the class should be immutable, or the fields that are used to calculate the hashcode should be made final.