Consider a health care application scenario, we need to add a list of patients to a collection and there should be a way to get the required patient object, which collection should we use and why?

We should go with any Map implemented class (if there is no specific requirement then HashMap). Map gives the option to get an object based on its identity. We can create HashMap with id as key and complete patient object as value, later it can be accessible by passing id to get method of Map.

When do we need to use objects as keys in Map?

When we have a requirement to store more than one field as key for Map. Consider we do not have an id field for any object, and more than 1 field is considered to uniquely identify that object.

If there is a Patient class with multiple fields, and patient's first name and ssn number is considered to uniquely identify that patient. How to store objects of this class in HashMap?

- Create Patient class
- Override equals and hashCode methods by considering first name and ssn field
- Set both key and value as patient object

Patient.java:

```
public class Patient {
	private String firstName;
	private String lastName;
	private String ssn;
	private int yearOfBirth;
	private String someOtherField;

	// No-arg and all-arg constructor
	// Override toString
	// Setter/Getter

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

```
	@Override
	public boolean equals(Object obj) {
		if (this == obj)
			return true;
		if (obj == null)
			return false;
		if (getClass() != obj.getClass())
			return false;
		Patient other = (Patient) obj;
		if (firstName == null) {
			if (other.firstName != null)
				return false;
		} else if (!firstName.equals(other.firstName))
			return false;
		if (ssn == null) {
			if (other.ssn != null)
				return false;
		} else if (!ssn.equals(other.ssn))
			return false;
		return true;
	}
}
```

Main.java:

```
public class Main {
	public static void main(String[] args) {
		Map<Patient, Patient> patientMap = new HashMap<>();

		Patient p1 = new Patient("Steve", "Smith", "123", 1990, "Some field 1");
		Patient p2 = new Patient("Joe", "Root", "456", 1980, "Some field 2");
		Patient p3 = new Patient("Jane", "Smith", "123", 2020, "Some field 3");

		patientMap.put(p1, p1);
		patientMap.put(p2, p2);
		patientMap.put(p3, p3);

		Patient newP2 = new Patient("Joe", "Root", "456", 0, null);
		System.out.println("Get Patient 2 details: " + patientMap.get(newP2));
	}
}
```

O/P:

Get Patient 2 details: Patient [firstName=Joe, lastName=Root, ssn=456, someOtherField=Some field 2]

Why do we need to override the hashCode method? What will happen if we do not override?

Default implementation of hashCode (from Object class) returns value by converting the object's internal address into an integer. So the hashcode of two similar objects will not be the same, and it will be impossible to get data for that object by calling the get method of generated HashMap.

With same above code - comment hashcode method
Patient newP2 = new Patient("Joe", "Root", "456", 1980, "Some field 2");
System.out.println("Get Patient 2 details: " + patientMap.get(newP2));
System.out.println("P2: " + p2.hashCode() + " .. New P2: " + newP2.hashCode());

O/P:
Get Patient 2 details: null
P2: 1550089733 .. New P2: 1442407170

Why do we need to override the equals method? What will happen if we override hashCode but not equals?

Getting value from HashMap is 3 step process

1. Get hashcode of passed key
2. Find required bucket based on hashcode
3. Find exact object in that bucket by comparing with equals method

Object class default “equals” method compares reference of objects. When we create a new object to get data from HashMap, its reference will be different then what is stored in HashMap, so again we won’t be able to get value from map.

Uncomment hashcode and comment equals method
Patient newP2 = new Patient("Joe", "Root", "456", 1980, "Some field 2");
System.out.println("Get Patient 2 details: " + patientMap.get(newP2));
System.out.println("P2: " + p2.hashCode() + " .. New P2: " + newP2.hashCode());
System.out.println("p2.equals(newP2): " + p2.equals(newP2));

O/P:
Get Patient 2 details: null
P2: 2366966 .. New P2: 2366966
p2.equals(newP2): false

Here though p2 and newP2 objects hashcode are same, Patient 2 details are null since “equals” method is returning false.

What is equals and hashcode contract?

Equal objects must have the same hashcode. Since in all hash related implementations, we rely on hash first, and then value, there is no point in overriding equals without

hashcode, if the first condition (comparing hash) is not passed, second condition (comparing values) is of no use.

What will happen if we override hashcode and return fixed value? Will it satisfy equals and hashcode contact?

```
@Override
public int hashCode() {
        return 1;
}
```

Though it satisfies the general contract between equals and hashcode, it has performance issues. Since all objects will have the same hashcode, all will be stored in a single bucket which may slow performance while retrieving data from map.

How is the bucket index calculated?

Below is the calculation to find bucket index:

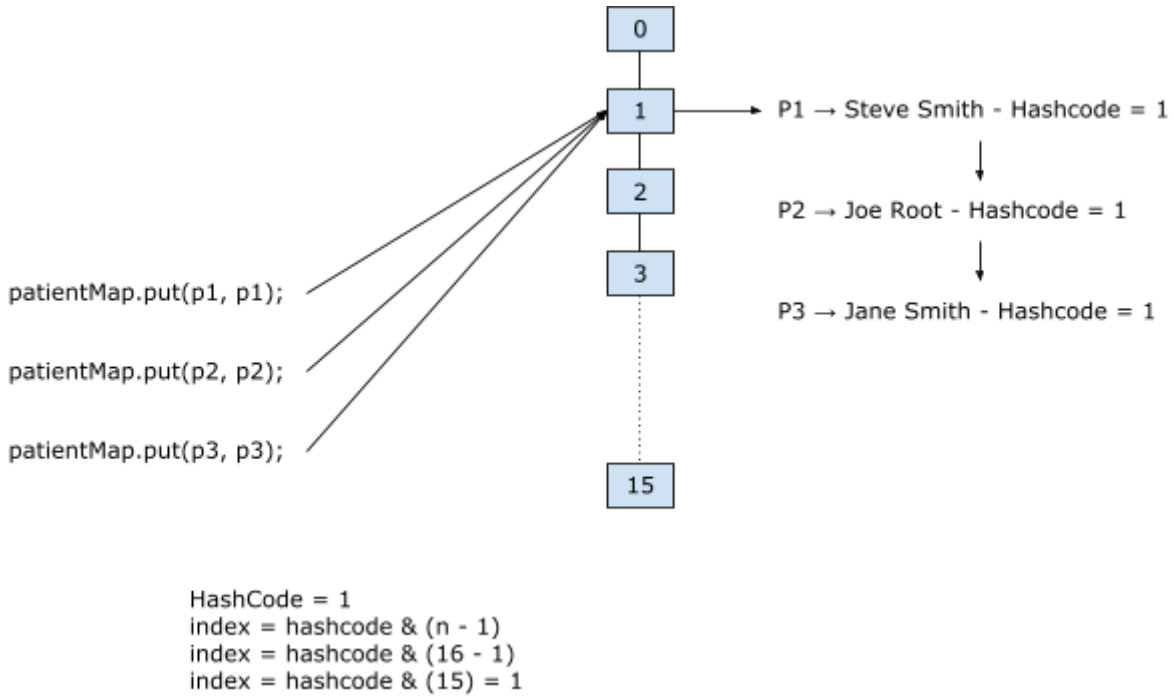
index = (n - 1) & hash
Here n is the capacity of HashMap.

```
System.out.println("P1 bucket: " + (p1.hashCode() & 15));
System.out.println("P2 bucket: " + (p2.hashCode() & 15));
System.out.println("P3 bucket: " + (p3.hashCode() & 15));
```

O/P:
P1 bucket: 0
P2 bucket: 6
P3 bucket: 5

Why will there be a performance impact when we pass a fixed value from hashCode()?

If we pass a fixed value from hashCode, all objects will get stored in the same bucket. While retrieving an element, it will always find a single bucket and will traverse to each node by comparing content with the equals method. Consider that we have 10000 objects then it will take a lot of time to get an object if it is there as last node.



How do objects get stored in a bucket? What will happen if we have 10k objects in a single bucket?

It follows the below steps when we perform the put operation.

- Check if bucket is empty:
    a. If Yes:
    Stores object as first node.
    b. If No, check if a new object already exists by comparing the "equals" method on existing objects.
        i. If Yes:
        Update existing object.
        ii. If No, check size of bucket:
            1. If less than or equal to 8:
            Prepare a linked list to add a new object as the next node of the last object.
            2. If more than 8:
            Prepare a balanced tree by copying existing data and use it to store objects.

Converting linked lists to balanced tree feature is added on Java8.
static final int TREEIFY_THRESHOLD = 8;

Why does it convert the linked list to balanced tree after threshold?

Linked list has a time complexity of O(n), which means that it works fine if n is a smaller number but n is a big number it takes time to get the result.

Balanced tree has a time complexity of O (log n) which is far better than O(n).
Below is the table to show the worst case scenario for linked list and balanced tree.

Number of records in bucket	Linked List - O(n)	Balanced Tree - O(log n)
5000	5000	13
10000	10000	14

Here we can see that if the number of records are 5000 in any bucket and the key exists as the last node or not available then the linked list loops 5000 times and the balanced tree only 13 times as it eliminates the search of half with every check.

What each node contains in a bucket?

Each node contains a hashcode, key, value and pointer to the next node.

HashMap has a table to store array of nodes
int hash
K key
V value
Node<K,V> next

Each index in the table is known as a bucket.

Key	Hashcode	Value	Pointer to next node

If there is no next node then the pointer to next node will be null.

Why can’t we use hashcode directly instead of bucket index?

Hashcode is usually a very big number (like 2366966), programmatically it is possible to add that many index but it will have lot of performance issue

---

What will happen if we store multiple objects with “null” as key?

Here “null” will be considered as a key, so the first object will insert successfully and then the second object will replace the first object as their keys are the same.

Patient p1 = new Patient("Steve", "Smith", "123", 1990, "Some field 1");

```
Patient p2 = new Patient("Joe", "Root", "456", 1980, "Some field 2");
patientMap.put(null, p1);
patientMap.put(null, p2);

System.out.println("Get Patient details: " + patientMap.get(null));
```

O/P:
Get Patient details: Patient [firstName=Joe, lastName=Root, ssn=456, yearOfBirth=1980, someOtherField=Some field 2]

In which bucket index map with "null" key be stored?

Bucket of the key with null will is always 0.

What is Collision in Map?

Multiple hash codes having the same bucket index is called collision.

What will be the capacity of HashMap if we insert 100 objects and why?

If using the default load factor (which is 0.75) then the capacity of HashMap will be 256.

Default initial capacity of HashMap is 16 with load factor 0.75. It means it accommodates 12 entries (16 * 0.75 = 12), when we add the 13th entry, it increases its size by 2, it checks and performs this operation with every put event.

Elements	Old Capacity	New Capacity	Reason
1 to 12	16	16	16 * 0.75 >= 12
13	16	32	16 * 0.75 < 13, Increase capacity by 2
25	32	64	32 * 0.75 < 25, Increase capacity by 2
49	64	128	64 * 0.75 < 49, Increase capacity by 2
97	128	256	128 * 0.75 < 97, Increase capacity by 2
100	256	256	256 * 0.75 >= 100

How to get HashMap capacity?

If we need to check only for our understanding purpose then debugging that map with any IDE would help. Eclipse provides good support to check capacity, bucket index, and other values. For some eclipse version it does not show all details.

If the requirement is to get capacity in Java code, then reflection can be used.

```
Map<String, String> map = new HashMap<>();
for (int index = 0; index < 100; index++) {
        map.put("Str " + index, "Some Value");
}
Field tableField = HashMap.class.getDeclaredField("table");
tableField.setAccessible(true);
Object[] table = (Object[]) tableField.get(map);
System.out.println("Capacity: " + (table == null ? 0 : table.length));
```

O/P:
Capacity: 256

What is the load factor? Can we specify our own load factor?

Load factor is a factor which indicates how many elements a HashMap can accommodate. If it reaches its threshold it increases its size by 2. It has a constructor which accepts initial size and load factor if there is a need to specify.

```
public HashMap(int initialCapacity, float loadFactor) {
...
```

We can specify our own load factor by using a parameterized constructor of HashMap.
`Map<String, String> map = new HashMap<>(100, 1);`

Can we have any number in the load factor or is there any restriction?

There is no restriction for load factor, but ideally it should be less than 1 for HashMap to perform better.

Why should the load factor be less than 1, what if we specify 10 as load factor?

We always want to make sure that the number of buckets are greater than number of entries in HashMap.

For 100 records, if we have more than 100 buckets then retrieval of elements will be fast.
With 10 as load factor, we will have 10 buckets for 100 records, this will result into Hash collision and retrieval process will get impacted.

If in the Patient table we have more than 10k records and there is a requirement to fetch Patient records from the database and create HashMap, what is the best approach performance wise?

If the number of records are already known, it is better to create HashMap with capacity which can accommodate all those records (in this case 10k), otherwise with every threshold limit, HashMap will create new copy with increased size and perform rehashing to store objects

---

How should we create HashMap to handle 10k objects?

Best way is to identify number to generate required capacity and use a parameterized constructor of HashMap to have a Map with that initial capacity.

```
public static int getNumForCapacity(int num) {
	return getNumForCapacity(num, 0.75f);
}

public static int getNumForCapacity(int num, float loadFactor) {
	float cap = num / loadFactor;
	int baseVal = 2;
	while (baseVal < cap) {
		baseVal *= 2;
	}
	return Math.round(baseVal * loadFactor);
}

…

In main method:
System.out.println("Capacity for 95 records: " + getNumForCapacity(95));
System.out.println("Capacity for 96 records: " + getNumForCapacity(96));
System.out.println("Capacity for 100 records: " + getNumForCapacity(100));
```

O/P:

```
Capacity for 95 records: 96
Capacity for 96 records: 96
Capacity for 100 records: 192

Create HashMap with initial capacity
Map<String, String> map = new HashMap<>(getNumForCapacity(10000));
```

What is the problem with directly passing initial capacity?

Default implementation of HashMap with initial capacity checks to next 2 to the power value and creates a table with that size, it does not consider load factor.

Passed Initial Capacity	Actual initial capacity
64	64
65	128
100	128
129	256

Consider example with 100 objects:

```
Map<String, String> map = new HashMap<>(100);
map.put("Test", "Test");
Field tableField = HashMap.class.getDeclaredField("table");
tableField.setAccessible(true);
Object[] table = (Object[]) tableField.get(map);
System.out.println("Capacity: " + (table == null ? 0 : table.length));
```

O/P:
Capacity: 128

This can accommodate 96 objects, the moment we add 97th objects, it increases its size to 256 and regenerates Map again.

```
Map<String, String> map = new HashMap<>(100);
for (int index = 1; index <= 96; index++) {
        map.put("Str " + index, "Some Value");
}
Field tableField = HashMap.class.getDeclaredField("table");
tableField.setAccessible(true);
Object[] table = (Object[]) tableField.get(map);
System.out.println("Capacity till 96th object: " + (table == null ? 0 : table.length));
map.put("Str 97", "Some Value");
table = (Object[]) tableField.get(map);
System.out.println("Capacity after 97th object: " + (table == null ? 0 : table.length));
```

O/P:
Capacity till 96th object: 128
Capacity after 97th object: 256

Here though we passed initial capacity as 100, by the time it reaches 100th objects it will have some other capacity.

What is rehashing? Why is it important?

Rehashing means hashing again, when HashMap increases its size after reaching threshold, it creates a new table with bigger capacity and copies each element from the previous table to the new table.

It is important to achieve time complexity of O(1), consider a scenario where rehashing is not there, when we insert 1000 elements, HashMap will try to accommodate it with default 16 size only. Most of the buckets may have around 50-70 elements, which means that all will have a balanced tree to store objects (as elements are greater than threshold of 8) all buckets which have time complexity of O(log n).

Does rehashing occur only when the overall threshold is crossed for HashMap? Or does it happen in any other case as well?

HashMap also considers size of buckets, if any single bucket reaches threshold, HashMap increases its size to double.

Return 1 from the hashcode method so all elements go to a single bucket and try below code.

```
for (int i=1; i<=8; i++) {
        Patient p = new Patient("First", "Last", i + "", 1990, "");
        patientMap.put(p, p);
}
Field tableField = HashMap.class.getDeclaredField("table");
tableField.setAccessible(true);
Object[] table = (Object[]) tableField.get(patientMap);
System.out.println("Capacity till 8th object in same bucket: " + (table == null ? 0 :
table.length));

Patient p = new Patient("First", "Last", 9 + "", 1990, "");
patientMap.put(p, p);

table = (Object[]) tableField.get(patientMap);
System.out.println("Capacity after 9th object: " + (table == null ? 0 : table.length));
```

O/P:
Capacity till 8th object in same bucket: 16
Capacity after 9th object: 32

If we use the Patient class object as the key, is there any problem with the already implemented Patient class?

Yes, the implemented Patient class is mutable, we can change values for all fields including those that are considered as key to store in HashMap.

If we make keys mutable then hashcode will not be consistent which can cause lookup failure.

```
Map<Patient, Patient> patientMap = new HashMap<>();

Patient p1 = new Patient("Steve", "Smith", "123", 1990, "Some field 1");
Patient p2 = new Patient("Joe", "Root", "456", 1980, "Some field 2");

patientMap.put(p1, p1);
patientMap.put(p2, p2);

Patient newP2 = new Patient("Joe", "Root", "456", 0, null);
System.out.println("Get Patient 2 details before key change: " +
patientMap.get(newP2));

p2.setFirstName("Joy");
System.out.println("Get Patient 2 details after key change: " + patientMap.get(newP2));
```

O/P:
Get Patient 2 details before key change: Patient [firstName=Joe, lastName=Root, ssn=456, yearOfBirth=1980, someOtherField=Some field 2]
Get Patient 2 details after key change: null

What will happen if we use String or Long (ID) as keys in HashMap? Will these creare similar issues?

String and all wrapper classes (Long, Byte, Integer, etc) are already immutable, so we can safely use them as keys to HashMap.

How to change Patient class to make it immutable?

Below are the steps to make immutable class:

1. Remove all setters - This ensures that fields cannot be modified one initialized.
2. Make all fields final and private - This ensures that fields cannot be modified outside the class. This step enforce to have all args constructor.
3. Make class as final - This ensures that subclass cannot modify override methods.
4. If a mutable object is used as reference, return a deep copy - This ensures that the original object is unchanged.

```
public final class Patient {

    private final String firstName;
    private final String lastName;
    private final String ssn;
    private final int yearOfBirth;
```

```
	private final String someOtherField;

	public Patient(String firstName, String lastName, String ssn, int yearOfBirth,
String someOtherField) {
		this.firstName = firstName;
		this.lastName = lastName;
		this.ssn = ssn;
		this.yearOfBirth = yearOfBirth;
		this.someOtherField = someOtherField;
	}

	// Override hashcode, equals and toString
	// Getters
}
```

If a class is referencing one mutable class, what will happen if we do not provide a deep copy with the getter method?

Though if the referencing class is not part of the key, HashMap will be able to lookup the record, but the class will be considered as a partially mutable class only.

Consider we have Address class as member variable of Patient class.

Address.java:

```
public class Address {
	private String country;
	public Address (String country) {
		this.country = country;
	}
	// Setter, Getter
}
```

Patient.java:

```
public final class Patient {
	private final String firstName;
	private final String lastName;
	private final String ssn;
	private final Address address;
	public Patient(String firstName, String lastName, String ssn, Address address) {
		this.firstName = firstName;
		this.lastName = lastName;
		this.ssn = ssn;
		this.address = address;
	}
	…
}
```

Main.java:

```
Patient p2 = new Patient("Joe", "Root", "456", new Address("England"));
System.out.println("P2 details before: " + p2);
Address address = p2.getAddress();
address.setCountry("India");
System.out.println("P2 details after: " + p2);
```

O/P:

```
P2 details before: Patient [firstName=Joe, lastName=Root, ssn=456, address=Address
[country=England]]
P2 details after: Patient [firstName=Joe, lastName=Root, ssn=456, address=Address
[country=India]]
```

What changes should be done to make the Patient class immutable?

With getter, return a deep copy instead of the original reference. There are two ways to do it:

1. Create a new address object in patient class and return.
   Patient.java:
   ```
   public Address getAddress() {
           return new Address(this.address.getCountry());
   }
   ```

2. Create a method in the Address class to provide a copy of passed object.