Core Java – 2021 – 2022

**Covariant, Contravariant and Invariant** **Covariance 🡪**

class Super {

Object getSomething() {}

}

**List<String> is a subtype of List<Object>** 🡪 **Covariant**

**List<Object> is a subtype of List<String>** 🡪 **Contravariant**

invariant that neither is a subtype of the other

class Sub extends Super {

String getSomething() {}

}

**Covariant**: a Cat[] is an Animal[] , **Contravariant**: an Animal[] is a Cat[]

Sub#getSomething is covariant because it returns a subclass of the return type of Super#getSomething (but fulfills the contract of Super.getSomething()).

**class** Parent {  
 Object get() {  
 **return new** Object();  
 }  
}

**class** Child **extends** Parent {  
 @Override  
 String get() {  
 **return new** String(**"Child"**);  
 }  
}

**Contravariance**

class Super {

void doSomething( **String parameter** )

}

class Sub extends Super {

void doSomething( **Object parameter** )

}

Sub#doSomething is contravariant because it takes a parameter of a superclass of the parameter of

Super#doSomething (but, again, fullfills the contract of Super#doSomething)

**Generics**

This is also possible for Generics:

List<String> aList...

List<? extends Object> **covariantList** = aList;

List<? super String> **contravariantList** = aList;

**Covariance = narrowing conversion.**

**Contravariance = widening conversion.**

**Invariance (in this context) = not convertible.**

**Get/Put Principle**

In summary, we **use covariance when we only intend to take generic values out of a structure**. We **use contravariance when we only intend to put generic values into a structure**

**Overriding Method is covariant.**

**class** Child **extends** Parent {  
 @Override  
 String get() {  
 **return new** String(**"Child"**);  
 }  
}

**class** Parent {  
 Object get() {  
 **return new** Object();  
 }  
}

**Static Binding VS Dynamic Binding**

**Binding** refers to the link between method call and method definition.

1. Static Binding (also known as Early Binding). When type of object is determined at compile time
2. Dynamic Binding (also known as Late Binding). When type of object is determined at run time

Static binding uses Type(Class in Java) information for binding while Dynamic binding uses Object to resolve binding. If there is any private, final or static method in a class, there is static binding.

**Example of static binding**

**class** Dog {

**private** **void** eat() {System.out.println("dog is eating...");}

**public** **static** **void** main(String args[]) {

Dog d1=**new** Dog();

d1.eat();

 }

}

**Example of dynamic binding**

**class** Animal {

**void** eat(){System.out.println("animal is eating...");}

}

**class** Dog **extends** Animal {

**void** eat() {System.out.println("dog is eating..."); }

**public** **static** **void** main(String args[]) {

Animal a=**new** Dog();

a.eat();  🡺 Output:dog is eating...

}

}

|  |  |
| --- | --- |
| **Static Binding** | **Dynamic Binding** |
| It is a binding that happens at compile time. | It is a binding that happens at run time. |
| Actual object is not used for binding. | Actual object is used for binding. |
| It is also called early binding because binding happens during compilation. | It is also called late binding because binding happens at run time. |
| Method overloading is the best example of static binding. | Method overriding is the best example of dynamic binding. |
| Private, static and final methods show static binding. Because, they can not be overridden. | Other than private, static and final methods show dynamic binding. Because, they can be overridden. |

**AutoCloseable – Try with Resouce**

We have a new super interface **[java.lang.AutoCloseable](http://docs.oracle.com/javase/7/docs/api/java/lang/AutoCloseable.html" \o "AutoCloseable)**. This interface have one method:

**void close() throws Exception;**

Java docs recommend this interface to be **implemented on any resource that must be closed when it is no longer needed**. When we open any such AutoCloseable resource in special try-with-resource block, immediately after finishing the try block, **JVM calls this close() method on all resources initialized in “try()” block**.

You can also implement the above interface for cleaning operations.

**public class** CustomResource **implements** AutoCloseable {  
 **public void** accessResource() {  
 System.***out***.println(**"Accessing the resource"**);  
 }  
   
 @Override  
 **public void** close() **throws** Exception {  
 System.***out***.println(**"CustomResource closed automatically"**);  
 }  
}

**public class** TryWithCustomResource {  
 **public static void** main(String[] args) {  
 **try** (CustomResource cr = **new** CustomResource()) {  
 cr.accessResource();  
 } **catch** (Exception e) {  
 e.printStackTrace();  
 }  
 }  
}

OUTPUT

Accessing the resource

CustomResource closed automatically

When we open any such AutoCloseable resource in special try-with-resource block, immediately after finishing the try block, **JVM calls this close() method on all resources initialized in “try()” block.**

**How HashMap Works in Java**

Hashing is the mechanism of assigning unique code to a variable or attribute using an algorithm to enable easy retrieval. A true hashing mechanism should always return the same hashCode() when it is applied to the same object. Then comes the question, "How does hashing help in storing and retrieving the value in HashMap?" Many will say that the value will be stored in the bucket and retrieved using the key. If you think that is how it works then you are absolutely wrong. To prove it, let's take a look at the HashMap class:

/\*\*

\* The table, resized as necessary. Length MUST Always be a power of two.

\*/

**transient Entry[] table;**

So what is the use of Entry[] in a HashMap for? Because the HashMap stores the Objects as **Entry instances,**not as**key and value.**

## What Is Entry Class?

HashMap has an inner class called an Entry Class which holds the key and values. And there is also something called next, which you will get to know a bit later.

static class Entry<K,V> implements Map.Entry<K,V> {

**final K key;**

**V value;**

**Entry<K,V> next;**

**final int hash;**

........

}

You know that the HashMap stores the Entry instances in an array and not as key-value pairs. In order to store a value, you will use the put() method of the HashMap, so now let's dig into that and see how it works.

## How Does Put() Method Work Internally?

The Code [**implementation of the put() method**](https://dzone.com/articles/www.javainterviewpoint.com/) will look like this:

public V put(K key, V value) {

if (key == null)

return putForNullKey(value);

int hash = hash(key.hashCode());

int i = indexFor(hash, table.length);

for (Entry<K,V> e = table[i]; e != null; e = e.next) {

Object k;

if (e.hash == hash && ((k = e.key) == key || key.equals(k))) {

V oldValue = e.value;

e.value = value;

e.recordAccess(this);

return oldValue;

}

}

modCount++;

addEntry(hash, key, value, i);

return null;

}

* First, it checks if the key given is null or not. If the given key is null, it will be stored in the zero position, as the hashcode of null will be zero.
* Then it applies the hashcode to the key **.hashCode()** by calling the hashcode method. In order to get the value within the limits of an array, the hash(key.hashCode()) is called, which performs some shifting operations on the hashcode.
* The **indexFor()** method is used to get the exact location to store the Entry object.
* Then comes the most important part what happens if two different object has the same hashcode( eg : Aa,BB will have the same hashcode) and will it be stored in the same bucket. To handle this let's think of the LinkedList in data structure it will have a next attribute which will always point to the next object. The same way the next attribute in the Entry class points to the next object. Using this different objects with the same hashcode will be placed next to each other.

In the case of the Collision, the HashMap checks for the value of the next attribute if it is **null** it inserts the Entry object in that location, if next attribute is not null then it keeps the loop running till next attribute is null then stores the Entry object there.

## **How are Duplicate Keys Prevented in HashMap?**

As we all know, HashMap doesn't allow duplicate keys, even though when we insert the same key with different values, only the latest value is returned.

import java.util.HashMap;

import java.util.Map;

public class HashMapEg {

public static void main(String[] args) {

Map map = new HashMap();

map.put(1,"sam");

map.put(1,"Ian");

map.put(1,"Scott");

map.put(null,"asdf");

System.out.println(map);

}

}

For the above code, you will get the output as  {null=asdf, 1=Scott} ,  as the values sam  and Ian  will be replaced by Scott. So, how does this happen?

All the Entry Objects in the LinkedList will have the same hashcode, but HashMap uses **equals()** . This method checks the equality, so if **key.equals(k)**is true, then it will replace the value object inside the Entry class and not the key. So this way it prevents the duplicate key from being inserted.

## **How Does Get() Method Work Internally?**

Almost the same logic as applied in the put() method will be used to retrieve the value.

public V get(Object key) {

if (key == null)

return getForNullKey();

int hash = hash(key.hashCode());

for (Entry<K,V> e = table[indexFor(hash, table.length)];e != null;e = e.next) {

Object k;

if (e.hash == hash && ((k = e.key) == key || key.equals(k)))

return e.value;

}

return null;

}

* First, it gets the hash code of the key object, which is passed, and finds the bucket location.
* If the correct bucket is found, it returns the value (e.value)
* If no match is found, it returns null.

## **What Happens If Two Keys Have the Same Hashcode?**

The same collision resolution mechanism will be used here. **key.equals(k)**will check until it is true, and if it is true, it returns the value of it.

# **Why Clone method is protected**

The Clonable interface is just a marker saying the class can support clone. The method is protected because you shouldn't call it on object, you can (and should) override it as public. In class Object, the clone() method is declared protected. If all you do is implement Cloneable, only subclasses and members of the same package will be able to invoke clone() on the object. To enable any class in any package to access the clone() method, you'll have to override it and declare it public.

**You don't always want your objects to be cloneable. Sometimes you have classes that represent something that should not be cloneable**. **Interface Cloneable is a marker interface (an interface without any methods - its only purpose is to indicate that the class that implements it is allowed to be cloned.**

Since subclasses inherit public instance methods and access to inherited methods cannot be decreased, if Object declared clone() as public, then every class would have to have a public clone() method. This would pollute the api of every class regardless of whether it could actually be cloned or not.

Citation From **Josh Bloch's Effective Java**:  
*"The Cloneable interface was intended as a mixin interface for objects to advertise that they permit cloning. Unfortunately it fails to serve this purpose ... This is a highly a typical use of interfaces and not one to be emulated ... In order for implementing the interface to have any effect on a class, it and all of its superclasses must obey a****fairly complex, unenforceable and largely undocumented protocol****"*