Core Java – 2024

**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() {Sop("dog is eating...");}

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

Dog d1=**new** Dog();

d1.eat();

 }

}

**Example of dynamic binding**

**class** Animal {

**void** eat() {Sop("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;**

**Code prior to Autoclosable interface**

BufferedReader br = **null**;  
 **try** {  
 String sCurrentLine;  
 br = **new** BufferedReader(**new** FileReader(**"C:/temp/test.txt"**));  
 **while** ((sCurrentLine = br.readLine()) != **null**) {  
 System.***out***.println(sCurrentLine);  
 }  
 } **catch** (IOException e) {  
 e.printStackTrace();  
 } **finally** {  
 **try** {  
 **if** (br != **null**) **br.close()**;  
 } **catch** (IOException ex) {  
 ex.printStackTrace();  
 }  
 }

**With Autoclosabe, new way of writing**

**try (BufferedReader br = new BufferedReader(new FileReader("C:/temp/test.txt")))** {  
 String sCurrentLine;  
 while ((sCurrentLine = br.readLine()) != null) {  
 System.out.println(sCurrentLine);  
 }  
 } catch (IOException e) {  
 e.printStackTrace();  
 }

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** MyClosbale **implements** AutoCloseable {  
  
 **public** String getSomeInfo() {  
 **return "some info"**;  
 }  
  
 @Override  
 **public void** close() **throws** Exception {  
 System.***out***.println(**"Getting closed ..."**);  
 }  
}

**public class** Test {  
 **public static void** main(String[] args) **throws** Exception {  
 **try**(MyClosbale cl = **new** MyClosbale()) {  
 System.***out***.println(cl.getSomeInfo());  
 }  
 }  
}

**OUTPUT**

some info

Getting closed ...

**public static void** main(String[] args) **throws** Exception {  
 MyClosbale cl = **new** MyClosbale();  
 System.***out***.println(cl.getSomeInfo());  
}

**OUTPUT**

some info

**\*\* In the above close method is never called, it means, it gets called when you provide try() block.**

But if you simply write like this,

You can also write like this.

**try (CustomResource cr = new CustomResource()) {**

**cr.accessResource();**

**}**

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.**

**What is a race condition**

**A race condition occurs when two or more threads can access shared data and they try to change it at the same time**. Problems often occur when one thread does a "check-then-act" (e.g. "check" if the value is X, then "act" to do something that depends on the value being X) and another thread does something to the value in between the "check" and the "act". In order to prevent race conditions from occurring, you would typically put a lock around the shared data to ensure only one thread can access the data at a time.

**How AtomicInteger or AtomicLong works in Java**

A compare and swap operation compares the contents of a memory location to a given value and, only if they are the same, modifies the contents of that memory location to a given new value. This is done as a single atomic operation.

AtomicInteger class provides few methods which perform the addition and subtraction operations **atomically**.

* addAndGet() – Atomically adds the given value to the current value and returns new value *after* the addition.
* getAndAdd() – Atomically adds the given value to the current value and returns old value.
* incrementAndGet() – Atomically increments the current value by 1 and returns new value *after* the increment. It is equivalent to **++i** operation.
* getAndIncrement() – Atomically increment the current value and returns old value. It is equivalent to **i++** operation.
* decrementAndGet() – Atomically decrements the current value by 1 and returns new value *after* the decrement. It is equivalent to **i- –** operation.
* getAndDecrement() – Atomically decrements the current value and returns old value. It is equivalent to **– -i** operation.

**It internally uses private static final** Unsafe ***U*** = Unsafe.*getUnsafe*();

**import** jdk.internal.misc.Unsafe;

**public final int** getAndSet(**int** newValue) {  
 **return *U***.getAndSetInt(**this**, ***VALUE***, newValue);  
}

**public final boolean** compareAndSet(**int** expectedValue, **int** newValue) {  
 **return *U***.compareAndSetInt(**this**, ***VALUE***, expectedValue, newValue);  
}

**public final int** getAndIncrement() {  
 **return *U***.getAndAddInt(**this**, ***VALUE***, 1);  
}

Composition over Inheritance

The reasons to prefer composition are profound.

**Inheritance**

* Use inheritance when you have a clear hierarchical relationship between classes, and you can say one class is a specialized form of another.
* **Inheritance is sometimes useful when hierarchy is more important than the formation of objects. It relates to Open-Closed Principle, which states that classes should be closed for modification but open to extension. That way you can have polymorphism**
* It’s suitable when you want to reuse code from the base class and when changes to the base class should propagate to derived classes. **Inheritance is a good choice when you need to override or extend the behavior of the base class**.
* There are two benefits of inheritance: **subtyping** and **subclassing**. Subtyping means conforming to a type (interface) signature, i.e. a set of APIs, and one can override part of the signature to achieve subtyping polymorphism. **Subclassing means implicit reuse of method implementations.**
* As a rule of thumb, I tend to choose inheritance over composition when polymorphic use is expected. For example, having a polymorphic class Widget in GUI frameworks, or a polymorphic class Node in XML libraries allows to have an API which is much more readable and intuitive to use
* You can't change the implementation inherited from super classes at runtime (obviously because inheritance is defined at compile time).

**Why Inheritance Breaks Encapsulation**

**Inheritance exposes protected members**. **This breaks encapsulation of the parent class**, and if used by subclass.

**Composition**

* If your object needs to appear as a different object or behave differently depending on an object state or conditions, then use Composition: Refer to State and Strategy Design Patterns. One example of this: **You want to create a Stack out of a List. Stack only has pop, push and peek. You shouldn't use inheritance given that you don't want push\_back, push\_front, removeAt, et al.-kind of functionality in a Stack**.
* **Use composition to combine simple objects into more complex ones**. It’s ideal when you want to model a has-a relationship rather than an is-a relationship.
* **Composition is preferable when you want to change the parts of a whole dynamically at runtime**, or when you want to limit the visibility of the composed objects’ methods and data.
* **Composition avoids a proliferation of classes.**
* **Delegation is one example of a way to use composition instead of inheritance. Delegation lets you modify the behavior of a class without subclassing**. It can be achieved using Decorator pattern. Composition provides pluggability.
* **Composition is a very simple and tactical way of building objects**. Using composition you can always choose to define your own behavior or simply expose that part of your composed parts

When you want to "Copy"/Expose the base class' API, you use inheritance. When you only want to "copy" functionality, use delegation.

**Inheritance creates tight, compile-time coupling between the classes whereas Composition in contrast is loose coupling, which among others enables clear separation of concerns, the possibility of switching dependencies at runtime and easier, more isolated dependency testability.**

Generics

# **Type-1 : Fully Generic Type**

**public class GenericQ1<T> {**  
 private LinkedList<T> list = new LinkedList<>();  
  
 **public void offer(T t) {  
 list.add(t);  
 }**  
 **public T poll() {  
 T t = list.removeFirst();  
 return t;  
 }**  
 @Override  
 public String toString() {  
 return list.toString();  
 }  
  
 public static void main(String[] args) {  
 GenericQ1<Person> personQ = new GenericQ1<>();  
  
 for (int i = 0; i < 5; i++) {  
 Person p = new Person("Name-" + i);  
 personQ.offer(p);  
 }  
  
 System.*out*.println("All Persons : " + personQ);  
 Person person1 = personQ.poll();  
 System.*out*.println("Removed Person :::" + person1);  
 }  
}

In the above case the class has been defined with generic type <T>. The structure is given below.

**public class GenericQ1<T> {**

}

# **Type – 2: Methods are generic**

Let use create a general class and create generic method. The code is given below.

public class GenericQ2 {

private LinkedList<Object> list = new LinkedList<>();

public <T> void offer(T t) {

list.add(t);

}

public <T> T poll() {

T t = (T) list.removeFirst();

return t;

}

@Override

public String toString() {

return list.toString();

}

public static void main(String[] args) {

GenericQ2 personQ = new GenericQ2();

for (int i = 0; i < 5; i++) {

Person p = new Person("Name-" + i);

personQ.offer(p);

}

System.out.println("All Persons : " + personQ);

Person person1 = personQ.poll();

System.out.println("Removed Person :::" + person1);

}

}

Now you can mark the above highlighted area. Here the class is not of type generic. Let us see the basic difference.

public class GenericQ2<T> public class GenericQ2

Always remember that , if the class is not generic type and if you are defining pure generic method, you have to define the method in the following manner.

**<accessModifier> <T> returnType(can be void) method() {**

**}**

Example is given below.

public **<T> void** offer(T t) { }

public **<T> T** poll() { }

Trick: First write the method with method return type, add <T> before

**How to create a Generic Type Class**

public class Gen<T> {  
  
 public void doSomething(T t) {  
 if(t instanceof String) {  
 System.*out*.println("It is a String");  
 } else {  
 System.*out*.println("Not...");  
 }  
 }  
  
 public T getSomething(T t) {  
 return t;  
 }  
  
 public T get(T t) {  
 **T[] ts = (T[]) Array.*newInstance*(t.getClass(), 5);**  
 ts[0] = t;  
 T t1 = ts[0];  
 return t;  
 }  
}

Test

public static void main(String[] args) {  
 Gen<String> gen = new Gen<String>();  
 gen.doSomething("abcd");  
 String val1 = gen.get("PQRS");  
 System.*out*.println(val1);  
 String value2 = gen.getSomething("Hati");  
 System.*out*.println(value2);  
}

**System.identityHashcode()**

So, hashCode() internally calls System.identityHashCode() ***as long as you are not overriding it in you class, if you override hashCode() it will call your implementation.***

|  |  |  |
| --- | --- | --- |
| **Expression** | **System.identityHashcode()** | **Hashcode()** |
| Object s1 = "abcd" | **1023892928** | **2987074** |
| Object s2 = **new** String("abcd"); | **363771819** | **2987074** |
| Object s3 = "abcd"; | **1023892928** | **2987074** |
| String ss1 = "abcd" | **1023892928** | **2987074** |
| String ss2 = **new** String("abcd"); | **2065951873** | **2987074** |
| String ss3 = "abcd"; | **1023892928** | **2987074** |
| String ss4 = ss3.intern(); | **1023892928** | **2987074** |
| String ss5 = ss2.intern(); | **1023892928** | **2987074** |
| Object os1 = 5; | **1791741888** | **5** |
| Object os2 = Integer.*valueOf*(5); | **1791741888** | **5** |
| Object os3 = 5; | **1791741888** | **5** |
| Person p = new Person (Without overriding equals and hashcod | **558638686** | **558638686** |
| Person p = new Person (After overriding equals and hashcod | **558638686** | **71751693** |

**Difference between System.identityHashCode() and Object hashcode()**

Object hashcode gives a specific code for an object regardless of whether it is overridden or not.

**System.identityHashCode() gives the one type of pointer to the address location of an object.** If you do not override the hascode for an object both Object hashcode and System.identityHashCode() will give the same value. Let us consider an example.

public class Person {  
 private String name;  
  
 public Person(String name) {  
 this.name = name;  
 }  
  
 get()/set() methods

}

public static void main(String[] args) {  
 Person p = new Person("Deb");  
 System.*out*.println("Hashcode: " + p.hashCode());  
 System.*out*.println("System.identityHashCode: " + System.*identityHashCode*(p));  
}

**Output**

Hashcode: 796684896

System.identityHashCode: 796684896

It means that in the above case the object Person has same object hashcode and identityhascode.

If you override the hashcode of the Person class, then identity hashcode and object hascode will be different.

public class Person {  
 private String name;  
  
 public Person(String name) {  
 this.name = name;  
 }  
  
get()/set() methods  
@Override  
 public int hashCode() {  
 return Objects.*hash*(name);  
 }  
}

public static void main(String[] args) {  
 Person p = new Person("Deb");  
 *// After overiding HashCode* System.*out*.println("Hashcode: " + p.hashCode());  
 System.*out*.println("System.identityHashCode: " + System.*identityHashCode*(p));  
}

**OUTPUT**

Hashcode: 68608

System.identityHashCode: 672320506

**It means two objects are equals according to equals method, they have the same object hascode but may not be the same identity hashcode**. **If two objects are equal according to reference equality ie ==, they will have the same identity hashcode**. It means identity hashcode is usefull which making object comparision using == or reference equality. **IdentityHashMap internally uses Syste.identityHashCode()** or identity hashcode.

**Why 31 in hashcode ?**

Essentially the prime value is used to reduce collisions. The Java code uses 31, therefore you should use a prime other than 31 for your own calculations, such as 37.

**How JVM Handles abstract classes**

Every object has a pointer to its **vtable** in its object header. The **vtable contains pointers to all virtual and abstract methods defined in the hierarchy of the type of the object**. They are ordered and have well-known indices which makes it performant to call such a method.