1 - Java Class Design

< Index Page | Next >

§1.1 Object class methods - equals/hashCode, toString

- If a class overrides the equals() method but not the hashCode() method, two objects will be considered equal while their hash code values will be different because Object class's hashCode method returns unique hashcode for every object. Because of the different hashcode, you cannot find a value using a key that is equal, but has different hashcode.
- When auto-unboxing will try to convert null into a primitive, it will throw a NullPointerException.

# Rules for the equals method:

- It is reflexive.
- It is symmetric.
- It is transitive.
- It is consistent.
- For any non-null reference value x, x.equals(null) should return false.

§1.2 Inheritance and Polymorphism

- The fundamental aspect of overriding is that it is the actual class of object and not the class of the reference type that determines which instance method will be invoked.
- Access to fields (and static methods) is bound at compile time and is never polymorphic. That is why if a field (or a static method) by the same name is defined in the base class as well as the derived class, it is class of the reference (and not the class of the actual object) that determines which field will be accessed. However if "this" is implicit the variable of the same class as method is used!
- You cannot have two methods of same signature in the same class. The return type is not considered.
- Java 5 onwards supports co-variant return types, which means that an overriding method can declare any sub-class of the return type declared in the overridden method as its return type. In case of primitives, the return type of the overriding method must be the same as that of the overridden method.
- Private means private to the class and not to the object.
- `PlaceHolder<String, String> ph2 = PlaceHolder<String>.getDuplicateHolder("b");` is not correct because getDuplicateHolder is a method and not a constructor. Therefore, you cannot specify generic type on the right hand side in this manner. You can do: `PlaceHolder<String, String> ph1 = PlaceHolder.<String>getDuplicateHolder("b");`
- Although, class B extends class A and `i` is a protected member of A, B still cannot access `i` through a reference of type A because A's `i` does not belong to B or, in other words, B is not involved in the implementation of A. B can access `i` through a reference of type B or its subclass.

Had the process() method been defined as `process(B b);` instead of `process(A a);` b.i would have been accessible as B is involved in the implementation of B.
- You will use composition when you are trying to reuse functionality from multiple classes and your class already extends from a framework class. Composition is a good way to reuse functionality from multiple class because it does not require your class to extend from any other class.

# Map:

- A Map is nothing but a set of buckets holding key-value pairs. Each bucket corresponds to a unique hashcode.
- When you store a key-value pair in a Map, the following things happen:
  1. Hashcode of the key is computed. This key is used to identify the bucket where the key-value must be stored.
  2. The key - value pair is stored in that bucket wrapped in a Map.Entry object.
  3. If multiple keys have same hashcode value, all those key-value pairs are stored in the same bucket.

- Now, a look up in a Map is a three step process:
  1. Hashcode of the key is computed. This code is used to identify the bucket where the key should be looked for.
  2. For all the key-value pairs in that bucket, check whether the supplied key is equal to the key in the bucket using equals() method.
  3. If a match exists, return the value, otherwise, return null.

- As you can see, it is critically important to make sure that hashCode() method return the same value for two objects that are equal as per equals() method. If this rule is not followed, you will not be able to retrieve the value from the map using another key object that is equal to the key object stored in the map.

§1.3 Singleton and Immutable classes

- All instances of wrapper classes and String class are immutable. The difference between StringBuffer and StringBuilder is that StringBuffer is thread safe while StringBuilder is not. Because of this StringBuilder is faster than StringBuffer.
- The Singleton patterns enforces that only one object of the class is ever created. To achieve this, you need to do 3 things:

  1. Make the constructor of the class private so that no one can instantiate it except this class itself.
  2. Add a private static variable of the same class to the class and instantiate it.
  3. Add a public static method (usually named getInstance()), that returns the class member created in step 2.

- To make a class immutable:
  1. Don't provide "setter" methods — methods that modify fields or objects referred to by fields.

2. Make all fields final and private.
3. Don't allow subclasses to override methods. The simplest way to do this is to declare the class as final. A more sophisticated approach is to make the constructor private and construct instances in factory methods.
4. If the instance fields include references to mutable objects, don't allow those objects to be changed:
    - Don't provide methods that modify the mutable objects.
    - Don't share references to the mutable objects. Never store references to external, mutable objects passed to the constructor; if necessary, create copies, and store references to the copies. Similarly, create copies of your internal mutable objects when necessary to avoid returning the originals in your methods.

§1.4 Use static keyword

- A static block will be executed only once when the class is loaded. A class is loaded when it is first referenced. In this case, it is first referenced by the JVM when it tries to run the main() method of this class.

< Index Page | Next >