Design Patterns

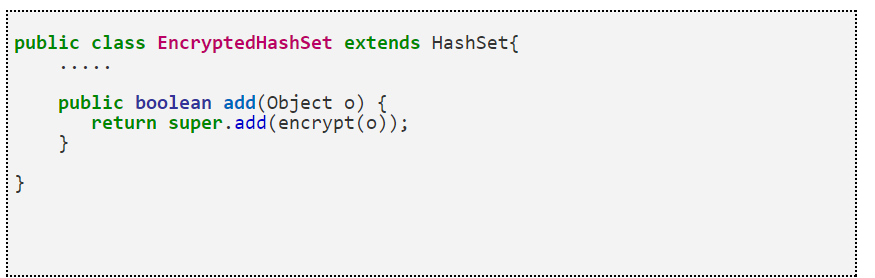
1. Constructor vs Static Factory method

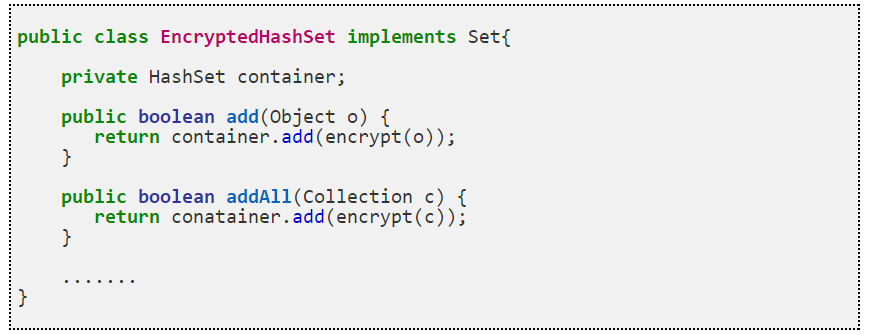
**Readable Names**  
One of the serious limitation of the constructor is that you cannot give it an explicit name, the name must be same as the name of the class. If your class return two different types of object for a different purpose, you can use [factory methods](http://javarevisited.blogspot.com/2015/06/difference-between-dependency-injection.html) with more readable names.  
  
A good example of this concept is java.text.NumberFormat class in JDK, which provides different factory methods to returns different objects e.g. getCurrencyInstance() to return an instance of NumberFormat which can format currency, getPercentInstance() to return a percentage format, and getNumberInstance() to return a general purpose number formatting.  
  
If you have used new NumberFormat(), it would have been difficult to know that which kind of [NumberFormat](http://www.java67.com/2015/06/how-to-format-numbers-in-java.html) instance would have returned.

**Polymorphism**  
Another serious limitation of a constructor is that it always return the same type of object. You cannot vary the type of constructed object, it must be same as the name of the contractor. But the factory method can return an object of different subclasses e.g. in above example, if you call getNumberInstance() then it return an object of [DecimalFormat](http://javarevisited.blogspot.sg/2012/03/how-to-format-decimal-number-in-java.html) class.  
  
The Polymorphism also allows static factory methods to return instances of non-public classes e.g. RegularEnumSet and JumboEnumSet in the case of EnumSet interface. When you call the EnumSet.of() method, which is a static factory method, it can return an instance of either of this class depending upon the number of enum constants in the Enum class you have provided.  
  
**Coupling**  
Factory methods promote the idea of coding using Interface then implementation which results in more flexible code, but constructor ties your code to a particular implementation.  
  
On the other hand by using constructor you tightly any client (who uses that class) to the class itself. Any change in class e.g. introducing a new constructor or new implementation will require change almost everywhere.  
  
Hence, it's very difficult to change and [unit test](http://javarevisited.blogspot.sg/2014/08/top-5-books-to-learn-unit-testing-junit-tdd-Java-programmers.html) the code which uses constructors to create objects all over.  
  
It's also one of the best practice in Java to make the constructor private to ensure that objects are only created using public static factory methods provided. The standard JDK API uses this technique with many new classes e.g. EnumSet.  
  
If you have ever used EnumSet then you might know that you cannot create instances of this class using a constructor, you must use the static factory method, EnumSet.of() to create an instance. This allows JDK to use choose the more relevant implementation of EnumSet depending upon the key sizes of provided Enum object.  
  
Disadvantages of static factory method

Like all things in the world, static factor methods also has some disadvantages e.g. once you make the constructor private to ensure that the class can only be instantiated using the constructor, you lost the power of inheritance because classes without public or protected constructor cannot be extended. But, if you look deep, this is not such a bad thing because it encourages you to [favor Composition over Inheritance](http://javarevisited.blogspot.sg/2013/06/why-favor-composition-over-inheritance-java-oops-design.html) for code reuse, which results in more robust and flexible software.  
  
In summary, both static factory methods and constructor have their usage and it's important for an experienced developer to understand their relative merits. In most cases, static factories are better choices so avoid the nature of providing public constructors without first considering static factories for creating objects.

1. Composition Over Inheritance

On composition, a class, which desire to use functionality of an existing class, doesn't inherit, instead it holds a reference of that class in a member variable, that’s why the name composition. Inheritance and composition relationships are also referred as IS-A and HAS-A relationships.  
  
1) One reason of favoring Composition over Inheritance in Java is fact that [Java doesn't support multiple inheritance](http://javarevisited.blogspot.com/2011/07/why-multiple-inheritances-are-not.html). Since you can only extend one class in Java, but if you need multiple functionality like e.g. for reading and writing character data into file, you need Reader and Writer functionality and having them as private members makes your job easy. That’s called composition. If you are following programming for interface than implementation principle, and using type of base class as member variable, you can use different Reader and Writer implementation at different situation. You won’t get this flexibility by using Inheritance, in case of extending a class, you only get facilities which are available at compile time.  
  
2) Composition offers better test-ability of a class than Inheritance. If one class is composed of another class, you can easily create [Mock Object](http://javarevisited.blogspot.sg/2014/04/difference-between-stub-and-mock-object-java-junit.html) representing composed class for sake of testing. Inheritance doesn't provide this luxury. In order to test derived class, you must need its super class. Since unit testing is one of the most important thing to consider during software development, especially in test driven development, composition wins over inheritance.  
  
3) Though both Composition and Inheritance allows you to reuse code, one of the disadvantage of Inheritance is that it breaks encapsulation. If sub class is depending on super class behavior for its operation, it suddenly becomes fragile. When behavior of super class changes, functionality in sub class may get broken, without any change on its part. One example of inheritance making code fragile is method add() and addAll() from [HashSet](http://javarevisited.blogspot.sg/2012/06/hashset-in-java-10-examples-programs.html). Suppose, If addAll() of HashSet is implemented by calling add() method and you write a sub class of HashSet, which encrypt the content before inserting into HashSet. Since there are only one methods add(), which can insert object into HashSet you override these method and called your encrypt() method by overriding add(). This automatically covers addAll() as well, because addAll() is implemented using add(), it looks very enticing.If you look closely you will see that this implementation is fragile, because its relied on super class behavior. If base class wants to improve performance and implements addAll() without calling add() method, following example will break.  


If you have *used Composition in favor of Inheritance* you won't face this problem and your class would have been more robust, because you are not relying on super class behavior any more. Instead you are using super class method for addition part and you will benefit with any improvement in addAll() as shown in below example:  
  


1. Another reason of favoring Composition over inheritance is flexibility. If you use Composition you are flexible enough to replace implementation of Composed class with better and improved version. One example is using [Comparator class](http://javarevisited.blogspot.com/2011/06/comparator-and-comparable-in-java.html) which provides compare functionality. if your Container object contains a Comparator instead of extending a particular Comparator for comparing , its easier to change the way comparison is performed by setting different type of Comparator to composed instance, while in case of inheritance you can only have one comparison behavior on runtime, You can not change it runtime.
2. Abstract class versus interface in the JDK 8

In general:

Interfaces are for when you want to say "I don't care how you do it, but here's what you need to get done."

Abstract classes are for when you want to say "I know what you should do, and I know how you should do it in some/many of the cases."

Java does not allow multiple inheritance but allows implementation of multiple interfaces.

Abstract classes allow non-static and non-final fields and allow methods to be public, private, or protected while interfaces' fields are inherently public, static, and final, and all interface methods are inherently public.

Abstract classes for behaviour that should not change, and is common for all subclasses.

1. Serialization in Java

Serialization is the proces of conversion of object’s state into sequence of bytes

1. Contract hashCode and equals

If hashcode value for 2 different objects is the same, it means that this can be the same object (equals can return true or false). However, if the hash code is different it means that equals will return false. This is the basis of whole Collection API and methods such as contains, hasKey(), order of sets etc.