Factory Pattern

```
public interface Product {
   public void operation();
}

public class ConcreteProduct implements Product {
   public void operation() {
      System.out.println("Product.operation() executed");
   }
}

public abstract class Creator {
   public abstract Product factoryMethod();
}

public class ConcreteCreator extends Creator {
   public Product factoryMethod() {
      return new ConcreteProduct();
   }
}

public class Client {
   public static void main(String[] args) {
      // create creator (strange ;) )
      Creator c = new ConcreteCreator();
      Product p = c.factoryMethod();  // use factory method to create product
      p.operation();  // use product
   }
}
```

Abstract Factory Pattern

```
public interface AbstractProductA {
   public void operationA();
}

public interface AbstractProductB {
   public void operationB();
}

public interface AbstractFactory {
   public AbstractProductA createProductA();
   public AbstractProductB createProductB();
}

public class ProductA1 implements AbstractProductA {
   public void operationA() {
      System.out.println("I am a ProductA1");
   }
}

public class ProductA2 implements AbstractProductA {
   public void operationA() {
      System.out.println("I am a ProductA2");
   }
}
```

```
public class ProductB1 implements AbstractProductB {
   public void operationB() {
      System.out.println("I am a ProductB1");
   }
}

public class ProductB2 implements AbstractProductB {
   public void operationB() {
      System.out.println("I am a ProductB2");
   }
}

public class ConcreteFactory1 implements AbstractFactory {
   public AbstractProductA createProductA() {
      return new ProductA1();
   }

   public AbstractProductB createProductB() {
      return new ProductB1();
   }
}

public class ConcreteFactory2 implements AbstractFactory {
   public AbstractProductA createProductA() {
      return new ProductA2();
   }

   public AbstractProductB createProductB() {
      return new ProductB2();
   }
}

public class Client {
   public static void main(String[] args) {
      // create factories
      AbstractFactory factoryOne = new ConcreteFactory1();
      AbstractFactory factoryTwo = new ConcreteFactory2();

      // use factories to create products
      AbstractProductA productA1 = factoryOne.createProductA();
      AbstractProductB productB1 = factoryOne.createProductB();

      AbstractProductA productA2 = factoryTwo.createProductA();
      AbstractProductB productB2 = factoryTwo.createProductB();

      // call methods on the products
      productA1.operationA();
      productB1.operationB();

      productA2.operationA();
      productB2.operationB();
   }
}
```

Singleton Pattern

```
public class Singleton {
   private static Singleton instance; // own instance

   /* protected to enable controlled subclassing */
   protected Singleton() {
   }

   public static Singleton getInstance() {

      // 'lazy' evaluate instance
      if (instance == null) {
         instance = new Singleton();
      }

      return instance;
   }

   public void operation() {
      System.out.println("Singleton.operation() executing" );
   }
}

public class Client {
   public static void main(String[] args) {
      // use getInstance to obtain Singleton instance
      Singleton s = Singleton.getInstance();

      // use operation
      s.operation();
   }
}
```

Lazy Singleton with Synchronised getInstance() Method

```
public class Singleton{
            private static Singleton uniqueInstance;

            private Singleton(){}

            public static synchronized Singleton getInstance(){
                        if (uniqueInstance == null){
                                    if (uniqueInstance == null){
                                                uniqueInstance = new Singleton();
                                    }
                        }

                        return uniqueInstance;
            }

}
```

Eager Singleton

```
public class Singleton{
        private static Singleton uniqueInstance = new Singleton();

        private Singleton(){}

        public static Singleton getInstance(){
                return uniqueInstance;
        }

}
```

Singleton with Double-Checked Locking

```
public class Singleton{
  /* Volatile keyword ensures that multiple threads handle the uniqueInstance
     variable correctly when it is being initialised to the singleton instance.
  */
  private volatile static Singleton uniqueInstance;

  private Singleton(){}

  public static Singleton getInstance(){
    //Check for an instance. If there isn't one, enter a synchronized block
    if (uniqueInstance == null){
      //We only need to synchronise the 1st time through the method
      synchronized(Singleton.class){
         //Once in the block, check again for instance. If still null, create an instance.
         if (uniqueInstance == null){
                uniqueInstance = new Singleton();
         }
      }
    }

    return uniqueInstance;
  }
}
```

Builder Pattern

```
public interface Product {
   public void operation();
}

public class ConcreteProduct implements Product {
   public ConcreteProduct() {
      System.out.println("constructing ConcreteProduct object");
   }

   public void operation() {
      System.out.println("ConcreteProduct.operation() executed");
   }
}
```

```
public interface Builder {
   public void buildPart();
   public Product getPart();
}

public class ConcreteBuilder implements Builder {
   Product p;

   public void buildPart() {
      p = new ConcreteProduct();
      // some more complex work with product
   }

   public Product getPart() {
      return p;
   }
}

public class Director {
   Builder build;

   public Director(Builder builder) {
      this.build = builder;
   }

   public void construct() {
      build.buildPart();
   }

}

public class Client {
   public static void main(String[] args) {
      // create builder
      Builder b = new ConcreteBuilder();

      // create director
      Director d = new Director(b);

      // construct, obtain and use
      d.construct();
      Product p = b.getPart();
      p.operation();
   }
}
```

Prototype Pattern

```
public interface Prototype {
   /** getClone() is used to seperate from Object's clone() method */
   public Prototype getClone();
   public void operation();
}

public class ConcretePrototype1 implements Prototype {
   public ConcretePrototype1() {
      System.out.println("constructing ConcretePrototype1");
   }

   public Prototype getClone() {
        // perform 'deep copy' if required
        return new ConcretePrototype1();
    }

    public void operation() {
        System.out.println("ConcretePrototype1.operation() executing");
    }
}

public class ConcretePrototype2 implements Prototype {
    public ConcretePrototype2() {
        System.out.println("constructing ConcretePrototype2");
    }

    public Prototype getClone() {
        // perform 'deep copy' if required
        return new ConcretePrototype1();
    }

    public void operation() {
        System.out.println("ConcretePrototype2.operation() executing");
    }
}

public class Client {
    public static void main(String[] args) {
        // create prototypical objects
        Prototype p1 = new ConcretePrototype1();
        Prototype p2 = new ConcretePrototype2();

        // generate objects from prototypical objects
        Prototype gp1 = p1.getClone();
        Prototype gp2 = p2.getClone();

        // call 'cloned' object's methods
        gp1.operation();
        gp2.operation();
    }
}
```