一、 创建模式之单例模式

1、 定义与特点

单例模式是一种对象创建型模式，使用单例模式，可以保证为一个类只生成唯一的实例对象。也就是说，在整个程序空间中，该类只存在一个实例对象。GoF 对单例模式的定义是:保证一个类、只有一个实例存在，同时提供能对该实例加以访问的全局访问方法。

2、 单例模式应用场景

在应用系统开发中，我们常常有以下需求:

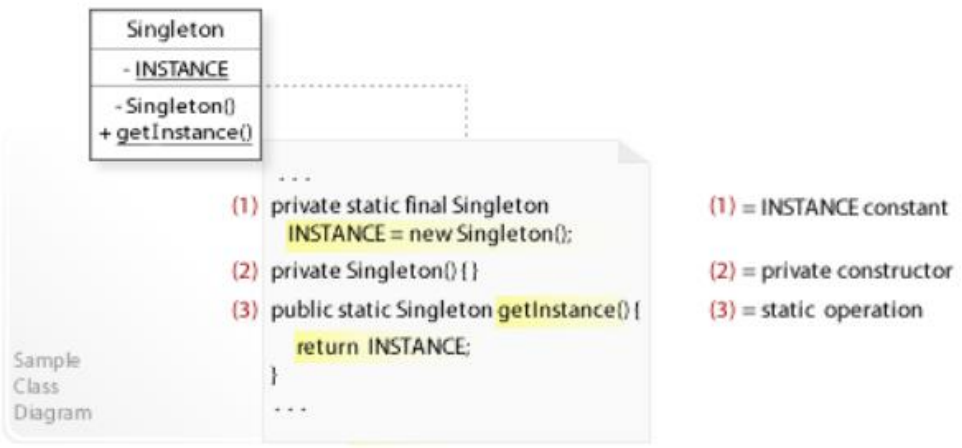
- 在多个线程之间，比如初始化一次 socket 资源;比如 servlet 环境，共享同一个资源或者操作同一个对象
- 在整个程序空间使用全局变量，共享资源
- 大规模系统中，为了性能的考虑，需要节省对象的创建时间等等。

因为单例模式可以保证为一个类只生成唯一的实例对象，所以这些情况，单例模式就派上用场了。

3、 单例模式实现步骤

1) 构造函数私有化
2) 提供一个全局的静态方法(全局访问点)
3) 在类中定义一个静态指针，指向本类的变量的静态变量指针

4、 示例



```
1  package com.sample.singleton.basic;
2  public class MyApp {
3      public static void main(String[] args) {
4          Singleton ref1 = null, ref2 = null;
5          ref1 = Singleton.getInstance();
6          ref2 = Singleton.getInstance();
7          if (ref1 == ref2) {
8              // The two singleton references are identical.
9              System.out.println("Singleton instantiated only once.");
10         }
11     }
12 }
```

```
Singleton instantiated only once.
```

```
1  package com.sample.singleton.basic;
2  public class Singleton {
3      // (1) INSTANCE constant that holds the sole instance.
4      private static final Singleton INSTANCE = new Singleton();
5      // (2) Private (hidden) constructor.
6      private Singleton() { }
7      // (3) Static operation that returns the sole instance.
8      public static Singleton getInstance() {
9          return INSTANCE;
10     }
11 }
```

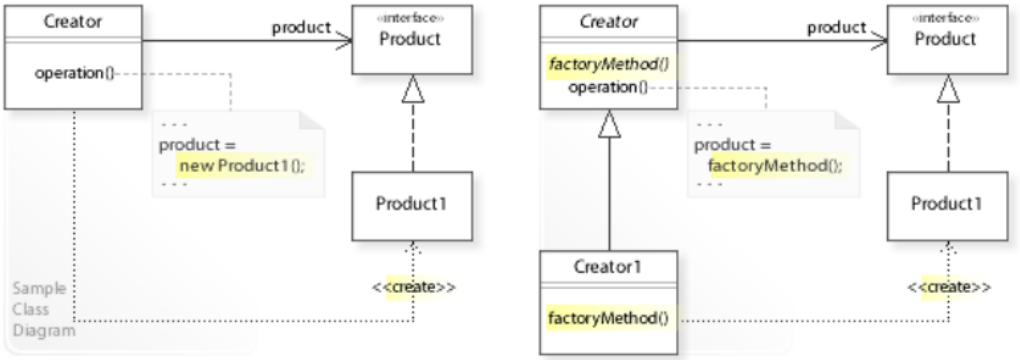
二、 创建模式之工厂模式

1、定义与特点

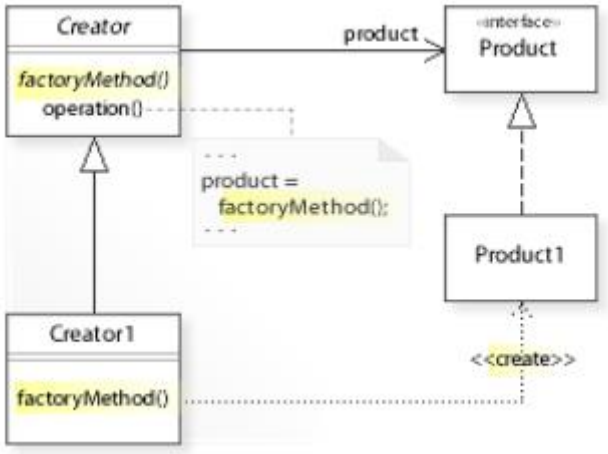
工厂方法模式同样属于类的创建模式，又被成为多态工厂模式，工厂模式的意义是定义一个创建产品的工厂接口，将创建工作推迟到子类中。核心工厂类不再负责产品的创建，这样，核心类成为一个抽象工厂角色，仅负责具体工厂子类必须实现的接口，好处是，可以在不修改具体工厂角色的情况下引入新的产品。

2、工厂模式组成部分

- 抽象工厂(Creator)角色
  抽象工厂模式的核心，包含对多个产品结构的声明，任何工厂类都必须实现这个接口。
- 具体工厂(Concrete Creator)角色
  具体工厂类是抽象工厂的一个实现，负责实例化某个产品族中的产品对象。
- 抽象(Product)角色
  抽象模式所创建的所有对象的父类，它负责描述所有实例所共有的公共接口。
- 具体产品(Concrete Product)角色
  抽象模式所创建的具体实例对象


Creator
operation()
product
«interface»
Product
product =
new Product1();
Product1
<<create>>
Sample
Class
Diagram
Creator
factoryMethod()
operation()
product
«interface»
Product
product =
factoryMethod();
Product1
Creator1
factoryMethod()
<<create>>

3、 示例


Creator
factoryMethod()
operation()
product
«interface»
Product
product =
factoryMethod();
Product1
Creator1
factoryMethod()
<<create>>

```
1  package com.sample.factorymethod.basic;
2  public class MyApp {
3      public static void main(String[] args) {
4          Creator creator = new Creator1();
5
6          System.out.println(creator.operation());
7      }
8  }
```

```
Hello World from Creator1!
Product1 created.
```

```
1  package com.sample.factorymethod.basic;
2  public abstract class Creator {
3      private Product product;
4
5      public abstract Product factoryMethod();
6
7      public String operation() {
8          product = factoryMethod();
9          return "Hello World from "
10                 + this.getClass().getSimpleName() + "!\n"
11                 + product.getName() + " created.";
12     }
13 }
```

```
1  package com.sample.factorymethod.basic;
2  public class Creator1 extends Creator {
3      public Product factoryMethod() {
4          return new Product1();
5      }
6  }
```

```
*********************************************************
Product inheritance hierarchy.
*********************************************************
```

```
1  package com.sample.factorymethod.basic;
2  public interface Product {
3      String getName();
4  }
```

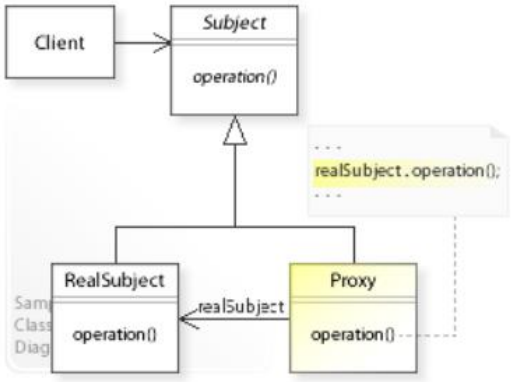
```
1  package com.sample.factorymethod.basic;
2  public class Product1 implements Product {
3      public String getName() {
4          return "Product1";
5      }
6  }
```

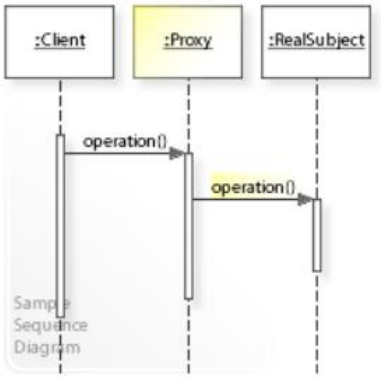
```
1  package com.sample.factorymethod.basic;
2  public class Product2 implements Product {
3      public String getName() {
4          return "Product2";



5      }
6  }
```

三、 结构型模式之代理模式

1、 定义与特点

代理模式是构造型的设计模式之一，它可以为其他对象提供一种代理(Proxy)以控制对这个对象的访问。所谓代理，是指具有与代理元(被代理的对象)具有相同的接口的类，客户端必须通过代理与被代理的目标类交互，而代理一般在交互的过程中(交互前后)，进行某些特别的处理。





2、 代理模式组成部分

- subject(抽象主题角色)

  真实主题与代理主题的共同接口。

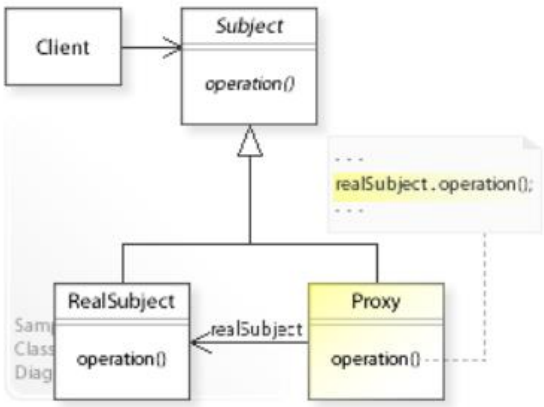
- RealSubject(真实主题角色)

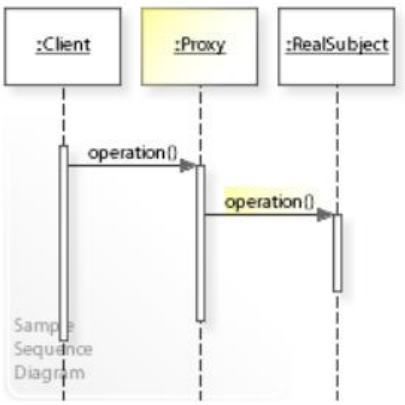
  定义了代理角色所代表的真实对象。

- Proxy(代理主题角色)

  含有对真实主题角色的引用，代理角色通常在将客户端调用传递给真是主题对象之前或者之后执行某些操作，而不是单纯返回真实的对象。

3、 示例





```java
package com.sample.proxy.basic;
public class Client {
    // Running the Client class as application.
    public static void main(String[] args) {
        // Creating a proxy for a real subject.
        Proxy proxy = new Proxy(new RealSubject());
        // Working through the proxy.
        System.out.println(proxy.operation());
    }
}
```

```
Hello world from Proxy and RealSubject!
```

```java
package com.sample.proxy.basic;
public abstract class Subject {
    public abstract String operation();
}
```

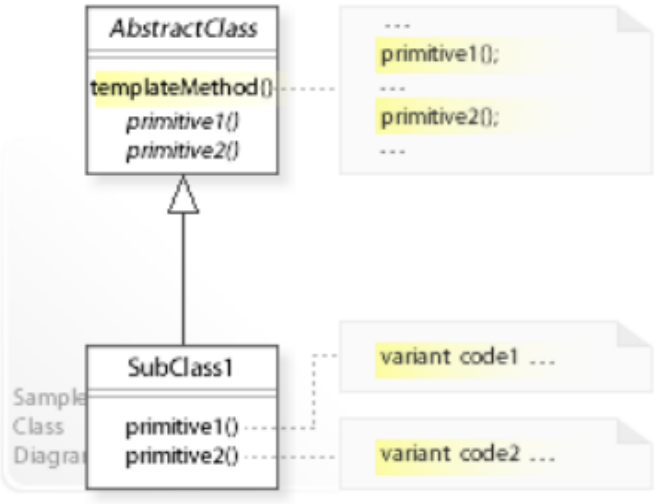
```java
package com.sample.proxy.basic;
public class RealSubject extends Subject {
    public String operation() {
        return "RealSubject!";
    }
}
```

```java
package com.sample.proxy.basic;
public class Proxy extends Subject {
    private RealSubject realSubject;

    public Proxy(RealSubject subject) {
        this.realSubject = subject;
    }
    public String operation() {
        return "Hello world from Proxy and " + realSubject.operation();
    }
}
```

四、 行为型模式之模板模式

1、定义与特点

模板方法模式是行为模式之一，他把具有特定步骤算法中的某些必要的处理委托让给抽象方法，通过子类继承对抽象方法的不同实现改变整个算法的行为。模板模式将某些步骤推迟到子类实现，并在操作过程中定义算法的框架。模板模式允许子类在不改变算法结构的情况下重新定义算法的某些步骤。



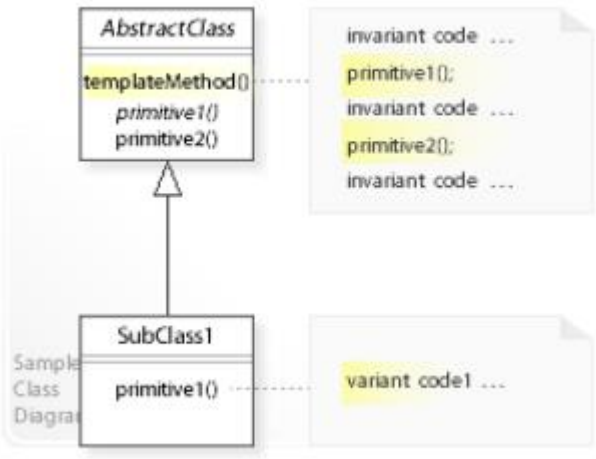
2、 模板模式应用场景

- 具有统一的操作步骤和操作过程
- 具有不同的操作细节
- 存在多个同样的操作步骤的应用场景，但某些具体操作细节却各不相同

3、 模板模式组成部分

- AbstractClass
  抽象类的父类
- ConcreteClass
  具体的实现子类
- templateMethod():
  模板方法
- method1()与 method2():
  具体步骤方法

4、 示例



```java
package com.sample.templatemethod.steps;
public abstract class AbstractClass {
    // Abstract primitive operation:
    // - provides no default implementation
    // - must be implemented (overridden).
    protected abstract void primitive1();
    //
    // Concrete primitive operation:
    // - provides a default implementation
    // - can be changed (overridden) optionally.
    protected void primitive2() {
        // variant code ...
    }
    public final void templateMethod() {
        // invariant code ...
        primitive1(); // calling primitive1 (variant code)
        // invariant code ...
        primitive2(); // calling primitive2 (variant code)
        // invariant code ...
    }
}
```

```java
package com.sample.templatemethod.steps;
public class SubClass1 extends AbstractClass {
    //
    protected void primitive1() {
        // variant code ...
    }
}
```