1、工厂方法模式（利用创建同一接口的不同实例）：

1、普通工厂模式：建立一个工厂类，对实现了同一接口的一些类进行实例的创建；

```
public class SendFactory {

    public Sender produce(String type) {
        if ("mail".equals(type)) {
            return new MailSender();
        } else if ("sms".equals(type)) {
            return new SmsSender();
        } else {
            System.out.println("请输入正确的类型!");
             return null;
         }
     }
}
```

2、多个工厂方法模式：提供多个工厂方法，分别创建对象；

```
public class SendFactory {

    public Sender produceMail(){
        return new MailSender();
    }

    public Sender produceSms(){
        return new SmsSender();
    }
}
```

3、静态工厂方法模式：将上面的多个工厂方法置为静态的，不需要创建工厂实例，直接调用即可；

4、适用场景：凡是出现了大量不同种类的产品需要创建，并且具有共同的接口时，可以通过工厂方法模式进行创建。在以上的三种模式中，第一种如果传入的字符串有误，不能正确创建对象，第三种相对于第二种，不需要实例化工厂类，所以，大多数情况下，我们会选用第三种—静态工厂方法模式。

2、抽象工厂模式（多个工厂）：创建多个工厂类，提高工厂的扩展性，不用像上面一样如果增加产品则要去修改唯一的工厂类；

3、单例模式（保证对象只有一个实例）：保证在一个JVM中，该对象只有一个实例存在；

1、适用场景：

1、某些类创建比较频繁，对于一些大型的对象，这是一笔很大的系统开销。

2、省去了new操作符，降低了系统内存的使用频率，减轻GC压力。

3、有些类如交易所的核心交易引擎，控制着交易流程，如果该类可以创建多个的话，系统完全乱了。（比如一个军队出现了多个司令员同时指挥，肯定会乱成一团），所以只有使用单例模式，才能保证核心交易服务器独立控制整个流程。

2、代码：

```
public class Singleton {

    /* 持有私有静态实例，防止被引用，此处赋值为null，目的是实现延迟加载 */
    private static Singleton instance = null;

    /* 私有构造方法，防止被实例化 */
    private Singleton() {
    }

     /* 静态工程方法，创建实例 */
     public static Singleton getInstance() {
         if (instance == null) {
             instance = new Singleton();
         }
         return instance;
```

```
    }

    /* 如果该对象被用于序列化，可以保证对象在序列化前后保持一致 */
    public Object readResolve() {
        return instance;
    }
}
```

3、分类：

1、饿汉式：类初始化时创建单例，线程安全，适用于单例占内存小的场景，否则推荐使用懒汉式延迟加载；

```
public class Singleton{
    private static Singleton instance = new Singleton();
    private Singleton(){}
    public static Singleton newInstance(){
        return instance;
    }
}
```

2、懒汉式：需要创建单例实例的时候再创建，需要考虑线程安全(性能不太好)：

```
public class Singleton{
    private static Singleton instance = null;
    private Singleton(){}
    public static synchronized Singleton newInstance(){
        if(null == instance){
            instance = new Singleton();
        }
        return instance;
    }
}
```

3、双重检验锁：效率高；(解决问题：假如两个线程A、B，A执行了if (instance == null)语句，它会认为单例对象没有创建，此时线程切到B也执行了同样的语句，B也认为单例对象没有创建，然后两个线程依次执行同步代码块，并分别创建了一个单例对象。)

```
public class Singleton {
    private static volatile Singleton instance = null;//volatile的一个语义是禁止指令重排序优化
    private Singleton(){}
    public static Singleton getInstance() {
        if (instance == null) {
            synchronized (Singleton.class) {
                if (instance == null) {//2
                    instance = new Singleton();
                }
            }
        }
        return instance;
    }
}
```

4、静态内部类方式：可以同时保证延迟加载和线程安全。

```
public class Singleton{
    private static class SingletonHolder{
        public static Singleton instance = new Singleton();
    }
    private Singleton(){}
    public static Singleton newInstance(){
        return SingletonHolder.instance;
    }
}
```

5、枚举：使用枚举除了线程安全和防止反射调用构造器之外，还提供了自动序列化机制，防止反序列化的时候创建新的对象。

```
public enum Singleton{
    instance;
    public void whateverMethod(){}
}
```

4、原型模式(对一个原型对象进行复制、克隆产生类似新对象)：将一个对象作为原型，对其进行复制、克隆，产生一个和元对象类似的新对象；

1、核心：它的核心是原型类Prototype，需要实现Cloneable接口，和重写Object类中的clone方法；

2、作用：使用原型模式创建对象比直接new一个对象在性能上要好的多，因为Object类的clone方法是一个本地方法，它直接操作内存中的二进制流，特别是复制大对象时，性能的差别非常明显。

5、适配器模式(接口兼容)：将某个类的接口转换成客户端期望的另一个接口表示，目的是消除由于接口不匹配所造成的类的兼容性问题。

1、类的适配器模式：

核心思想就是：有一个Source类，拥有一个方法，待适配，目标接口时Targetable，通过Adapter类，将Source的功能扩展到Targetable里，看代码：

```java
public class Source {

    public void method1() {
        System.out.println("this is original method!");
    }
}
```

```java
public interface Targetable {

    /* 与原类中的方法相同 */
    public void method1();

    /* 新类的方法 */
    public void method2();
}
```

```java
public class Adapter extends Source implements Targetable {

    @Override
    public void method2() {
        System.out.println("this is the targetable method!");
    }
}
```

2、对象的适配器模式：

只需要修改Adapter类的源码即可：

```java
public class Wrapper implements Targetable {

    private Source source;

    public Wrapper(Source source){
        super();
        this.source = source;
    }
    @Override
    public void method2() {
        System.out.println("this is the targetable method!");
    }

    @Override
    public void method1() {
        source.method1();
    }
}
```

3、接口的适配器模式：

这个很好理解，在实际开发中，我们也常会遇到这种接口中定义了太多的方法，以致于有时我们在一些实现类中并不是都需要。看代码：

```java
public interface Sourceable {

    public void method1();
    public void method2();
}
```

抽象类Wrapper2：

```java
public abstract class Wrapper2 implements Sourceable{

    public void method1(){}
    public void method2(){}
}
```

```java
public class SourceSub1 extends Wrapper2 {
    public void method1(){
        System.out.println("the sourceable interface's first Sub1!");
    }
}
```

```java
public class SourceSub2 extends Wrapper2 {
    public void method2(){
        System.out.println("the sourceable interface's second Sub2!");
    }
}
```

4、使用场景：

1、类的适配器模式：当希望将一个类转换成满足另一个新接口的类时，可以使用类的适配器模式，创建一个新类，继承原有的类，实现新的接口即可。

2、对象的适配器模式：当希望将一个对象转换成满足另一个新接口的对象时，可以创建一个Wrapper类，持有原类的一个实例，在Wrapper类的方法中，调用实例的方法就行。

3、接口的适配器模式：当不希望实现一个接口中所有的方法时，可以创建一个抽象类Wrapper，实现所有方法，我们写别的类的时候，继承抽象类即可。

6、装饰模式（给对象动态增加新功能，需持有对象实例）：装饰模式就是给一个对象增加一些新的功能，而且是动态的，要求装饰对象和被装饰对象实现同一个接口，装饰对象持有被装饰对象的实例：

1、示例：

Source类是被装饰类，Decorator类是一个装饰类，可以为Source类动态的添加一些功能，代码如下：

```java
public interface Sourceable {
    public void method();
}
```

```java
public class Source implements Sourceable {

    @Override
    public void method() {
        System.out.println("the original method!");
    }
}
```

```java
public class Decorator implements Sourceable {

    private Sourceable source;

    public Decorator(Sourceable source){
        super();
        this.source = source;
    }
    @Override
    public void method() {
        System.out.println("before decorator!");
        source.method();
        System.out.println("after decorator!");
    }
}
```

2、使用场景：

1、需要扩展一个类的功能。

2、动态的为一个对象增加功能，而且还能动态撤销。（继承不能做到这一点，继承的功能是静态的，不能动态增删。）

7、代理模式（持有被代理类的实例，进行操作前后控制）：采用一个代理类调用原有的方法，且对产生的结果进行控制。

```java
public interface Sourceable {
    public void method();
}
```

```java
public class Source implements Sourceable {

    @Override
    public void method() {
        System.out.println("the original method!
    }
}
```

```java
public class Proxy implements Sourceable {

    private Source source;
    public Proxy(){
        super();
        this.source = new Source();
    }
    @Override
    public void method() {
        before();
        source.method();
        atfer();
    }
    private void atfer() {
        System.out.println("after proxy!");
    }
    private void before() {
        System.out.println("before proxy!");
    }
}
```

8、**外观模式**(集合所有操作到一个类)：外观模式是为了解决类与类之间的依赖关系的，像spring一样，可以将类和类之间的关系配置到配置文件中，而外观模式就是将他们的关系放在一个Facade类中，降低了类类之间的耦合度。

```java
public class Computer {
    private CPU cpu;
    private Memory memory;
    private Disk disk;

    public Computer(){
        cpu = new CPU();
        memory = new Memory();
        disk = new Disk();
    }

    public void startup(){
        System.out.println("start the computer!");
        cpu.startup();
        memory.startup();
        disk.startup();
        System.out.println("start computer finished!");
    }

    public void shutdown(){
        System.out.println("begin to close the computer!");
        cpu.shutdown();
        memory.shutdown();
        disk.shutdown();
        System.out.println("computer closed!");
    }
}
```

9、桥接模式(数据库驱动桥接)： 桥接模式就是把事物和其具体实现分开，使他们可以各自独立的变化。桥接的用意是：将抽象化与实现化解耦，使得二者可以独立变化，像我们常用的JDBC桥DriverManager一样，JDBC进行连接数据库的时候，在各个数据库之间进行切换，基本不需要动太多的代码，甚至丝毫不用动，原因就是JDBC提供统一接口，每个数据库提供各自的实现，用一个叫做数据库驱动的程序来桥接就行了。

10、组合模式(部分整体模式)： 组合模式有时又叫部分-整体模式在处理类似树形结构的问题时比较方便。

11、享元模式(共享池、数据库连接池)： 享元模式的主要目的是实现对象的共享，即共享池，当系统中对象多的时候可以减少内存的开销，通常与工厂模式一起使用。当一个客户端请求时，工厂需要检查当前对象池中是否有符合条件的对象，如果有，就返回已经存在的对象，如果没有，则创建一个新对象，如数据库连接池；

12、策略模式(多种算法封装)： 策略模式定义了一系列算法，并将每个算法封装起来，使他们可以相互替换，且算法的变化不会影响到使用算法的客户。需要设计一个接口，为一系列实现类提供统一的方法，多个实现类实现该接口：

```
ICalculator cal = new Plus(); //ICalculator是统一接口，Plus是实现类(多个)
int result = cal.calculate(exp);  //jvm根据实现类不同而调用不同实现类的方法
```

13、模板方法模式(抽象方法作为骨架，具体逻辑让子类实现)： 定义一个操作中算法的框架，而将一些步骤延迟到子类中，使得子类可以不改变算法的结构即可重定义该算法中的某些特定步骤。完成公共动作和特殊动作的分离。

```
//题目：排序并打印：
abstract class AbstractSort {
    /**
     * 将数组array由小到大排序
     * @param array
     */
    protected abstract void sort(int[] array);

    public void showSortResult(int[] array){
        System.out.print("排序结果：");//打印
    }
}
//排序
class ConcreteSort extends AbstractSort {

    @Override
    protected void sort(int[] array){
        for(int i=0; i<array.length-1; i++){
            selectSort(array, i);
        }
    }

    private void selectSort(int[] array, int index) {
        //排序的实现逻辑
    }
}
//测试
public class Client {
    public static int[] a = { 10, 32, 1, 9, 5, 7, 12, 0, 4, 3 }; // 预设数据数组
    public static void main(String[] args){
        AbstractSort s = new ConcreteSort();
        s.showSortResult(a);
    }
}
```

14、观察者模式(发布-订阅模式)： 当一个对象变化时，其它依赖该对象的对象都会收到通知，并且随着变化！对象之间是一种一对多的关系。类似于邮件订阅和RSS订阅，当你订阅了该文章，如果后续有更新，会及时通知你。

15、迭代器模式(遍历集合)： 迭代器模式就是顺序访问聚集中的对象。

16、责任链模式(多任务形成一条链，请求在链上传递)： 有多个对象，每个对象持有对下一个对象的引用，这样就会形成一条链，请求在这条链上传递，直到某一对象决定处理该请求。但是发出者并不清楚到底最终那个对象会处理该请求，所以，责任链模式可以实现，在隐瞒客户端的情况下，对系统进行动态的调整。

17、命令模式(实现请求和执行的解耦)： 命令模式的目的就是达到命令的发出者和执行者之间解耦，实现请求和执行分开，熟悉Struts的同学应该知道，Struts其实就是一种将请求和呈现分离的技术，其中必然涉及命令模式的思想！

18、备忘录模式(保存和恢复对象状态)： 主要目的是保存一个对象的某个状态，以便在适当的时候恢复对象。

19、状态模式(对象状态改变时改变其行为)： 当对象的状态改变时，同时改变其行为。状态模式就两点：1、可以通过改变状态来获得不同的行为。2、你的好友能同时看到你的变化。

```
/**
 * 状态模式的切换类    2012-12-1
 * @author erqing
 *
 */
public class Context {

    private State state;

    public Context(State state) {
        this.state = state;
    }

    public State getState() {
        return state;
    }

    public void setState(State state) {
        this.state = state;
    }

    public void method() {
        if (state.getValue().equals("state1")) {
            state.method1();
        } else if (state.getValue().equals("state2")) {
            state.method2();
        }
    }
}
```

20、访问者模式(数据接口稳定，但算法易变)： 访问者模式把数据结构和作用于结构上的操作解耦合，使得操作集合可相对自由地演化。访问者模式适用于数据结构相对稳定算法又易变化的系统。因为访问者模式使得算法操作增加变得容易。访问者模式就是一种分离对象数据结构与行为的方法，通过这种分离，可达到为一个被访问者动态添加新的操作而无需做其它的修改的效果。

21、中介者模式： 中介者模式也是用来降低类类之间的耦合的。如果使用中介者模式，只需关心和Mediator类的关系，具体类类之间的关系及调度交给Mediator就行，这有点像spring容器的作用。

22、解释器模式(对于一些固定文法构建一个解释句子的解释器，如正则表达式)：解释器模式用来做各种各样的解释器，如正则表达式等的解释器。

23、建造者模式(创建复合对象)：工厂类模式提供的是创建单个类的模式，而建造者模式则是将各种产品集中起来进行管理，用来创建复合对象，所谓复合对象就是指某个类具有不同的属性

24、设计模式的六大原则：

1、开闭原则（Open Close Principle）

开闭原则就是说对扩展开放，对修改关闭。在程序需要进行拓展的时候，不能去修改原有的代码，实现一个热插拔的效果。所以一句话概括就是：为了使程序的扩展性好，易于维护和升级。想要达到这样的效果，我们需要使用接口和抽象类，后面的具体设计中我们会提到这点。

2、里氏代换原则（Liskov Substitution Principle）

里氏代换原则(Liskov Substitution Principle LSP)面向对象设计的基本原则之一。 里氏代换原则中说，任何基类可以出现的地方，子类一定可以出现。 LSP是继承复用的基石，只有当衍生类可以替换掉基类，软件单位的功能不受到影响时，基类才能真正被复用，而衍生类也能够在基类的基础上增加新的行为。里氏代换原则是对“开-闭”原则的补充。实现“开-闭”原则的关键步骤就是抽象化。而基类与子类的继承关系就是抽象化的具体实现，所以里氏代换原则是对实现抽象化的具体步骤的规范。—— From Baidu 百科

3、依赖倒转原则（Dependence Inversion Principle）

这个是开闭原则的基础，具体内容：真对接口编程，依赖于抽象而不依赖于具体。

4、接口隔离原则（Interface Segregation Principle）

这个原则的意思是：使用多个隔离的接口，比使用单个接口要好。还是一个降低类之间的耦合度的意思，从这儿我们看出，

其实设计模式就是一个软件的设计思想，从大型软件架构出发，为了升级和维护方便。所以上文中多次出现：降低依赖，降低耦合。

5、迪米特法则（最少知道原则）（Demeter Principle）

为什么叫最少知道原则，就是说：一个实体应当尽量少的与其他实体之间发生相互作用，使得系统功能模块相对独立。

6、合成复用原则（Composite Reuse Principle）

原则是尽量使用合成/聚合的方式，而不是使用继承

25、jdk中的设计模式：

1．单例模式：

- java.lang.Runtime#getRuntime()
- java.awt.Desktop#getDesktop()
- java.lang.System#getSecurityManager()

2．责任链模式：

- java.util.logging.Logger#log()
- javax.servlet.Filter#doFilter()

3．观察者模式：

- java.util.Observer/ java.util.Observable（很少在现实世界中使用）
- 所有实现java.util.EventListener（因此实际上各地的Swing）
- javax.servlet.http.HttpSessionBindingListener
- javax.servlet.http.HttpSessionAttributeListener
- javax.faces.event.PhaseListener

26、spring中的设计模式：

a. 简单工厂：spring中的BeanFactory就是简单工厂模式的体现，根据传入一个唯一的标识来获得bean对象，但是否是在传入参数后创建还是传入参数前创建这个要根据具体情况来定。

b. 单例模式：Spring下默认的bean均为singleton。

c. 代理模式：为其他对象提供一种代理以控制对这个对象的访问。 从结构上来看和Decorator模式类似，但Proxy是控制，更像是一种对功能的限制，而Decorator是增加职责。 spring的Proxy模式在aop中有体现，比如JdkDynamicAopProxy和Cglib2AopProxy。

d. 观察者模式：定义对象间的一种一对多的依赖关系，当一个对象的状态发生改变时，所有依赖于它的对象都得到通知并被自动更新。spring中Observer模式常用的地方是listener的实现。如ApplicationListener。

27、