1.单例模式

为什么要有单例模式

实际编程应用场景中，有一些对象其实我们只需要一个，比如线程池对象、缓存、系统全局配置对象等。这样可以就保证一个在全局使用的类不被频繁地创建与销毁，节省系统资源。

实现单例模式的几个要点

1. 首先要确保全局只有一个类的实例。

   要保证这一点，至少类的构造器要私有化。

2. 单例的类只能自己创建自己的实例。

因为，构造器私有了，但是还要有一个实例，只能自己创建咯！

3. 单例类必须能够提供自己的唯一实例给其他类

就是要有一个公共的方法能返回该单例类的唯一实例。

全局有且仅有一个，且其他人不能创建该对象，但是其他对象可以获取创建的实例

单例模式的6种实现

1.饿汉式—静态常量方式（线程安全）

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

加载类的时候创建实例，线程安全

2.饿汉式—静态常量方式(静态代码块)（线程安全）

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

和方式一原理一样，只不过通过静态代码块来创建实例，也是在加载类的时候创建

3.懒汉式（线程不安全）

```
public class Singleton {
    private static Singleton singleton;
    private Singleton() {}
    public static Singleton getInstance() {
        if (singleton == null) {
            singleton = new Singleton();
        }
        return singleton;
    }
}
```

其他对象获取该单例的时候创建，所以线程不安全（调用时初始化）

4、懒汉式（线程安全，方法上加同步锁）

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

因为锁的原因,效率会降低

5、双重校验锁（线程安全，效率高）

```
public class Singleton {
        private volatile static Singleton singleton;
        private Singleton() {}
        public static Singleton getSingleton() {
                if (singleton == null) {
                        synchronized (Singleton.class) {
                                if (singleton == null) {
                                                singleton = new Singleton();
                                }
                        }
                }
                return singleton;
        }
}
```

通过同步代码块来实现线程安全,volatile关键字保证各个线程对于静态对象实例的可见性

6.静态内部类实现(线程安全,效率高)

```
public class Singleton {
    private static class SingletonHolder {
        private static final Singleton INSTANCE = new Singleton();
    }
    private Singleton (){}
    public static final Singleton getInstance() {
        return SingletonHolder.INSTANCE;
    }
}
```

这种方式下 Singleton 类被装载了，instance 不一定被初始化。因为 SingletonHolder 类没有被主动使用，只有通过显式调用 getInstance 方法时，才会显式装载 SingletonHolder 类，从而实例化 instance。

注意内部类SingletonHolder要用static修饰且其中的静态变量INSTANCE必须是final的。

2.重载 null问题

重载函数调用时精确性的问题

```
public class Confusing {
    private Confusing(Object o) {
        System.out.println("Object");
    }
    private Confusing(double[] dArray) {
        System.out.println("double array");
    }
    public static void main(String[] args) {
        new Confusing(null);
    }
}
```

构造器和重载在执行时会选取的方法或构造器中选取最精确的一个

要想用一个null参数来调用 Confusing(Object)构造器，你需要这样写代码：new Confusing((Object)null)。这可以确保只有Confusing(Object)是可应用的。更一般地讲，要想强制要求编译器选择一个精确的重载版本，需要将实际的参数转型为形式参数所声明的类型。

```
public void testOverLoad(Integer i) {
    System.out.println("Int");
}

public void testOverLoad(String s) {
    System.out.println("String");
}

public static void main(String[] args) {
    //当2个形参不属于继承关系的时候,null就报错,因为没有最精确的一个了
    //必须显式申明类型
    confusing.testOverLoad((Integer) null);
    confusing.testOverLoad((String) null);
}
```

3.浮点数精度丢失问题

计算机是二进制的。浮点数没有办法是用二进制进行精确表示。我们的CPU表示浮点数由两个部分组成：指数和尾数，这样的表示方法一般都会失去一定的精确度，有些浮点数运算也会产生一定的误差。如：2.4的二进制表示并非就是精确的2.4。反而最为接近的二进制表示是 2.3999999999999999。浮点数的值实际上是由一个特定的数学公式计算得到的。

浮点数的计算

https://blog.csdn.net/weixin_44364444/article/details/105606265

4.排序算法比较

排序算法	平均时间复杂度	最好情况	最坏情况	空间复杂度	排序方式	稳定性
冒泡排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	In-place	稳定
选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	In-place	不稳定
插入排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	In-place	稳定
希尔排序	$O(n \log n)$	$O(n \log^2 n)$	$O(n \log^2 n)$	$O(1)$	In-place	不稳定
归并排序	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$	Out-place	稳定
快速排序	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(\log n)$	In-place	不稳定
堆排序	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(1)$	In-place	不稳定
计数排序	$O(n + k)$	$O(n + k)$	$O(n + k)$	$O(k)$	Out-place	稳定
桶排序	$O(n + k)$	$O(n + k)$	$O(n^2)$	$O(n + k)$	Out-place	稳定
基数排序	$O(n \times k)$	$O(n \times k)$	$O(n \times k)$	$O(n + k)$	Out-place	稳定

5. 32位系统和64位系统的区别

1. 运行能力不同。64位可以一次性处理8个字节的数据量，而32位一次性只可以处理4个字节的数据量，因此64位比32位的运行能力提高了一倍。
2. 内存寻址不同。64位最大寻址空间为2的64次方，理论值直接达到了16TB，而32位的最大寻址空间为2的32次方，为4GB，换而言之，就是说32位系统的处理器最大

只支持到4G内存，而64位系统最大支持的内存高达亿位数。

3. 运行软件不同。由于32位和64位CPU的指令集是不同的。所以需要区分32位和64位版本的软件。

PriorityQueue o1-o2是大顶堆

6.检查素数方法:

至于检查素数，这里用的是常见的 $O(\sqrt{N})$ 复杂度的算法来检查是不是素数，即检查小于 $\sqrt{N}$ 的数中有没有能整除 N 的。

```
public boolean isPrime(int N) {
        if (N < 2) return false;
        int R = (int) Math.sqrt(N);
        for (int d = 2; d <= R; ++d)
            if (N % d == 0) return false;
        return true;
    }
```