Java 基础知识点总结

1、Java 的优势

1）万物皆对象，所有的类最终都继承自 Object，这对多态以及通用 API 的编写带来了很大的方便。
2）垃圾处理机制，不用关心内存泄漏问题。
3）类可以实现多个接口，弥补了不能多继承的缺陷。
4）类库中的方法也比较完善，命名也更加合理。
5）语言本身开源，并且有很多开源的框架。
6）异常处理机制更加合理。
7）跨平台的，一次编译处处运行。

2、public 类和非 public 类

在一个.java 源文件中，只能有一个类可以用 public 修饰（内部类除外，可以将内部类看做外部类的特殊的成员，它可以用 public、protected、private 等任何可以修饰成员的修饰符修饰），该类必须和源文件同名。在同一个包中，public 类和非 public 类没有任何区别，main 函数可以放在任何一个类中，并且同一个包可以包含多个.java 源文件，也就是可以包含多个 public 类。在不同包之间，一个包中只能访问另一个包的 public 类，非 public 类是不可见的。

3、Java 中的数值类型

1）为什么 byte a = 1; 不报错，而 float b = 0.3; 报错？
解答：（1）整数在 Java 中默认为 int，浮点数默认为 double。
（2）整数在 Java 中的存储精度相同，而浮点数在 Java 中存储精度不同：
① byte a = 4; int b = 4; a == b（正确）
② float a = 4; double b = 4; a == b（正确）
③ float a = 0.3f; double b = 0.3; a != b（正确）
（3）编译器能够将 byte、short、char 类型范围内的整数自动转化为对应类型，但是不能将 float 类型范围呢的浮点数自动转化为 float（注意，编译器不能将 int 类型的变量自动转化为 byte、short、char 类型）：
① byte a = 100;（正确）
② int a = 100; byte b = a;（报错）
（4）编译器会将 byte、short 类型的计算上升为 int 进行计算：
① byte a = 3, b = 4; byte c = a + b;（报错，a + b 会上升为 int 进行计算，结果为 int）
② int a = 3, b = 4; int c = a + b;（正确，结果依旧为 int，当结果超过 int 范围时会变为负数）
（5）总结：byte 4 == int 4; float 0.4 != double 0.4; 。
2）为什么 lone a = 11111L; 最后要加上 L？

解答：当定义 long 类型变量时，如果所赋的整数没有超过 int 范围加不加 L 都可以，因为 int → long 不会损失精度；但是一旦超过，因为整数默认为 int 类型，所以不加 L 会报错。

3）Java 中的隐式类型转化：

byte、short、char → int → long → float → double（float 是用科学计数法存储，所以虽然 float 为 4 字节，long 为 8 字节，但是 float 的范围大于 long）

4）Java 中的所有数值类型都有正负。

5）二进制以 0b 开头，八进制以 0 开头，十六进制以 0x 开头。（数字 0）

4、初始化

1）初始化顺序

显示初始化和初始化块按顺序执行，均发生在构造器之前。

初始化块的作用：将多个构造器中的相同部分提取出来，实现了代码的复用。

```
class Test
{
    int i = 1; //显示初始化
    {
        i = 2; //初始化块，按顺序执行，i == 2
    }
    public Test()
    {
        i = 3; //构造器最后执行，i == 3
    }
}
```

2）静态初始化

一个类只有在首次使用（如实例化一个对象、调用该类的静态方法等）时才会被加载，此时会进行静态初始化。

3）常量的初始化

final 常量可以显示初始化，也可以在构造器中对 final 常量赋值，因此，一个类中的 final 域可以根据对象而有所不同。方法的形参可设为 final。

4）默认初始化

成员变量会进行默认初始化，而局部变量不会进行默认初始化，所以局部变量未进行初始化或赋值不可执行读取操作。数组对象的成员可以看做该数组类型的成员变量，所以是会进行

初始化的。

5）构造器

构造器中调用另一个构造器或者调用父类的构造器都只能写在第一句，也就是说构造器中的第一句要么是 this(…) 要么是 super(…)，如果没有明确写出的话，会默认加上一句 super()。

5、多态

1）重写方法不能使用比被重写方法更严格的访问权限，也不能抛出比被重写方法更宽泛的异常，这是因为多态。
2）多态是针对运行时发生的，任何域访问操作都将由编译器解析，因此不是多态的。静态方法是与类而非对象相关联的，因此也不会发生多态。

6、常用类

1）包装类

（1）自动装箱时的缓存问题：

```
Integer a1 = 100;
Integer b1 = 100;
Integer a2 = 200;
Integer b2 = 200;
System.out.println(a1 == b1); //true
System.out.println(a2 == b2); //false
```

自动装箱拆箱是编译器所做的优化，对于“Integer a1 = 100”，编译器会自动调用 Integer.valueOf(100)，查看 valueOf 方法的源码：

```
public static Integer valueOf(int i) {
    if (i >= IntegerCache.low && i <= IntegerCache.high)
        return IntegerCache.cache[i + (-IntegerCache.low)];
    return new Integer(i);
}
```

发现如果 i 在 [low,high] 范围内时，会直接从缓存里面取出一个事先 new 好的对象返回，这里 low，high 默认值分别为-128, 127，即 i 如果落在[-127, 128]，就会直接返回缓存里面的对象。

除了 Integer，其它包装类型对缓存的使用如下：
* Boolean 有两个缓存值 TRUE，FALSE
* Byte 表示范围是[-128,127]，全部值都是缓存的

* Short, Long 缓存范围是[-128,127]
* Float,Double 没有缓存
（2）字符串与基本数据类型、包装类之间转换：
基本数据类型、包装类 → String：调用 String.valueOf(Xxx x) 方法。
String →基本数据类型、包装类：调用包装类的 Xxx.parseXxx(String str) 方法。

2）时间、日期类

（1）System.currentTimeMillis()
返回当前时间的 long 类型值。此 long 值是从 1970 年 1 月 1 日 0 点 0 分 00 秒开始到当前的毫秒数。
（2）java.util.Date 类
Date d = new Date(); //返回当前时间的 Date
Date d1 = new Date(15231512541241L); //返回 long 值对应的日期
d. getTime(); //返回当前日期对应的 long 值
（3）字符串和时间对象的转换
借助 java.text.SimpleDateFormat 类来完成：
*在构造时传入一个格式化字符串或者使用默认格式化字符串
*Date → String 使用 SimpleDateFormat 的 format()方法
*String → Date 使用 SimpleDateFormat 的 parse()方法
示例代码：

```
// Date → String
SimpleDateFormat sdf = new SimpleDateFormat();
String date = sdf.format(new Date());
System.out.println(date);  // 17-9-15 下午 11:32

SimpleDateFormat sdf1 = new SimpleDateFormat("EEE, d MMM yyyy HH:mm:ss Z");
date = sdf1.format(new Date());
System.out.println(date);  //星期五, 15 九月 2017 23:32:45 +0800

// String → Date
Date date1 = sdf.parse("17-9-15 下午 11:32");
System.out.println(date1);

date1 = sdf1.parse("星期五, 15 九月 2017 23:32:45 +0800");
System.out.println(date1);
```

（4）java.util.Calendar 类
计算机认为 Date 是一个 long 类型的值，而人们认为日期时间应该是某年某月某日几点几分几秒，Calendar 类在二者之间起桥梁作用。
获取实例：Calendar c = Calendar.getInstance();
常用方法：get()/set()/add()/date getTime()/setTime()
示例代码：

```
Calendar c = Calendar.getInstance();
```

```
// c.set(2017, Calendar.SEPTEMBER, Calendar.FRIDAY, 23, 47, 0);
c.set(Calendar.YEAR, 2017);
c.set(Calendar.MONTH, Calendar.SEPTEMBER);
c.set(Calendar.DATE, Calendar.FRIDAY);

Date d = c.getTime();
System.out.println(d);

c.add(Calendar.MONTH, -3);
System.out.println(c.get(Calendar.MONTH));
```

3）String 类

（1）String 类对象一个常量对象，在创建后不能再修改，需要修改的话只能重新创建一个新的对象。
（2）字符串---->字节数组:调用字符串的 getBytes()；
字符串---->字符数组：调用字符串的 toCharArray()；
（3）StringBuffer 与 StringBuilder：
都是可变的字符序列；
StringBuffer 线程安全、效率低，StringBuilder 线程不安全、效率高；
一般使用 StringBuilder，因为该对象的线程安全应该由用户（程序员）来控制。

7、标签

Java 中的标签只能使用在循环语句之前，主要用于多层循环。

```
lable:
while (...)
{
    for (...;...;...)
    {
        ...
        continue lable; //直接转到 lable 处，并继续循环
        ...
        break lable; //直接转到 lable 处，但不重新进入循环
    }
}
```

8、接口

1）接口与接口之间是继承关系，而且可以多继承。
2）接口不继承自 Object 类

9、泛型

1）起因：使用多态的话会导致数据类型不明确，装入数据的类型都被当做 Object 对待，从而丢失了自己的实际类型。并且重写获取数据时往往需要转回原来的类型，效率低，而且容易产生错误。
注意：泛型仅仅是给编译器使用的，一旦编译完成，所有和泛型有关的类型全部被擦除。
2）声明时使用泛型，使用时指定类型，只能为引用类型。静态属性和静态方法不能使用泛型类所定义的字母，但是静态方法可以为泛型方法。没有泛型数组。

```
class Test<T1>
{
	T1 t1;
	//T1[] t2 = new T1[2]; //错误
	//static void func1(T1 t) {}	//错误
	static <T2> void func2(T2 t) {} //正确
	public static void main(String[] args)
	{
		//Test<String>[] a = new Test<String>[4]; //错误
	}
}
```

3）使用时如果不指定类型，则当做 Object 对待 。
4）在泛型中，List<Object> 与 List<String> 是并列关系，不是继承关系。
5）通配符：?
只能用于声明类型或声明方法，不能用在类上。

```
//class Test<?> {} //错误
class Test
{
	List<?> list = new ArrayList<String>(); //正确
	void func(List<?> list) {} //正确
}
```

List<?> 相当于 List<A>、List<B> 的父类，A、B 是具体类型。
List<? extends A>：可以存放 A 及其子类，即：? ≤ A；
List<? super A>：可以存放 A 及其父类，即：? ≥ A。

10、函数参数问题

1）可变参数

可变参数在方法中可以当做数组使用，最大的好处是传入参数时不用将参数封装成一个数组。

2）函数传参的方式

只有值传递一种方式。

11、重载和重写

1）区别

重载：方法名相同，参数列表不同，与返回值无关。
重写：同名同参同返回，前面的权限修饰符不能比父类更封闭，抛出的异常不能比父类更宽泛。

2）运算符重载

Java 中不支持运算符重载，因为运算符重载只是一种成员方法的语法糖，没有它并不会降低语言的性能，反而有时候会引起逻辑混乱，违反了 Java 语言的设计初衷。

3）equals() 和 hashcode() 方法重载

如果要重载的话这两个方法应该一起重载，并保证 equals 返回 true 的情况下，hashCode 返回值相同。

12、异常处理

1）具体异常类型是在运行时决定：

```
public class TestException
{
    public static int division(int a, int b) throws Exception
    {
        return a / b;
    }
    public static void main(String[] args)
    {
        int ret = 0;
        try
        {
            ret = division(1, 0);
        } catch (ArithmeticException e)
        {
            System.out.println("ArithmeticException");
```

```
        } catch (Exception e)
        {
            System.out.println("Exception");
        }
        System.out.println(ret);
    }
}
```

由于 division 方法抛出 Exception 类型的异常，所以在编译时强制需要有 catch (Exception e) 语句，但是运行时按实际抛出异常类型处理，所以结果为“ArithmeticException”。

2）运行时异常（RuntimeException）如：ArithmeticException、NullPointException、IndexOutOfBoundsException 等是程序的逻辑错误，也就是 bug，所以在编译时并不强制我们处理这类型异常，这些异常是需要我们修改代码来解决的，并不应该通过异常处理来解决，当然非要捕获处理这类异常也是可以的。
3）当异常被 catch 语句捕获并进行处理后，之后的代码会顺序执行并不会在处理完异常后直接退出。
4）只有发生程序终止（比如调用 System.exit(-1)）的情况下 finally 语句才不会执行，其他情况下 finally 语句一定会执行，比如当代码运行到 return 语句时会先执行 finally 语句再返回，而这种情况下 try-catch-finally 语句块之后的代码就不会执行了，所以要将资源清理操作放到 finally 语句块中。

13、位运算符

<<：左移，右边补 0，左移一位相当于乘以 2
>>：带符号右移，左边补符号位，右移一位相当于除以 2
>>>：不带符号右移，左边补 0

数	x	x<<2	x>>2	x>>>2
17	00010001	00|01000100	00000100|01	00000100|01
-17	11101111	11|10111100	11111011|11	00111011|11

例：将整数 110 从右端开始的 4-7 位变为 0

```
public class Test1
{
    public static void main(String[] args)
    {
        int a = 110;
        System.out.println(Integer.toBinaryString(a));
        int b = 0b1111 << 3;
        a = a & (~b);
        System.out.println(Integer.toBinaryString(a));
```

```
    }
}
```

14、常量

1）既可以在声明时直接初始化，也可以在初始化块中初始化，还可以在构造器中初始化。
2）为了节约内存，常量一般声明为静态的，此时只能在声明时直接初始化或者在静态初始化块中初始化。

15、内部类

1）内部类产生的字节码文件名为：Outer$Inner.class。
2）**可以将内部类看做外部类的一个成员**，可以用 public、protected、private、final、abstract、static 等修饰。非静态内部类的成员能够访问外部类的所有成员。非静态内部类中不能声明静态成员，静态内部类中可以声明。

```
class Outer
{
    private int id = 100;

    class Inner
    {
        int id = 50;
        void print()
        {
            int id = 25;
            System.out.println(id); //25
            System.out.println(this.id); //50
            System.out.println(Outer.this.id); //100
        }
    }

    void print()
    {
        Inner inner = new Inner();
        inner.print();
    }
}

class Test
{
    public static void main(String[] args)
    {
        Outer outer = new Outer();
```

```
        outer.print();
    }
}
```

3）创建一个非静态内部类的对象时必须绑定一个外部类对象。

```
class Test
{
    public static void main(String[] args)
    {
        Outer outer = new Outer();
        Outer.Inner inner = outer.new Inner();
        inner.print();
    }
}
```

4）静态内部类的成员不能访问外部类的非静态成员，创建时也不需要绑定外部类对象。

```
Outer.Inner inner = new Outer.Inner();
```

5）内部类还可以定义在成员方法、语句块中，此时若内部类要访问成员方法的参数或者局部变量，则必须将参数或局部变量定义为 final（原因为：**局部变量的生命周期与方法内部类对象的生命周期不一致**，局部变量在栈中，方法调用结束时就会被销毁，但是方法内部类对象在堆中创建，方法调用结束时，该对象可能任然存在并且可能任然在执行，此时局部变量已经被销毁，而 final 常量还在方法区中，所以方法内部类要访问成员方法的参数或者局部变量，则必须将参数或局部变量定义为 final）。

```
class Outer
{
    private int id = 100;

    void func(final int a)
    {
        final int b = 10;
        if (true)
        {
            {
                class Inner
                {
                    int id = 50;
                    void print()
                    {
                        int id = 25;
                        System.out.println(id); //25
                        System.out.println(this.id); //50
                        System.out.println(Outer.this.id); //100
                        System.out.println(b); //10
```

```
                    System.out.println(a); //5
                }
            }
            new Inner().print();
        }
    }
}

class Test
{
    public static void main(String[] args)
    {
        Outer outer = new Outer();
        outer.func(5);
    }
}
```

6）外部类的成员方法能够创建内部类对象，并访问内部类的私有成员（**注：一个类的私有成员在整个顶层类的范围内可见**）。比如单例模式中的登记式：

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

    private Singleton(){}

    public static final Singleton getInstance()
    {
        return SingletonHolder.INSTANCE;
    }
}
```

16、集合类

1）List

List 接口实现了 Collection 接口，并增加了一些有关索引的方法。有两个主要的子类：ArrayList 和 LinkedList。ArrayList 底层采用数组实现，LinkedList 是采用双向链表实现的。
Arrays 中的静态方法 asList（返回一个大小固定的 List）以及 ArrayList 中的成员方法 toArray 实现了数组和 List 之间的转换。

2）Set

Set 容器中的元素最好不要修改，因为当添加完成后修改元素可能会造成在 Set 容器中出现重复元素。

3）Hashtable

（1）与 HashMap 的区别

①主要：Hashtable 线程安全、同步、效率相对较低

HashMap 线程不安全、非同步、效率相对较高

②Hashtable 的父类是 Dictionary，而 HashMap 的父类是 AbstractMap。

③Hashtable 的键和值均不能为 null，HashMap 的键最多有一个为 null，值可以由多个。

④一般使用 HashMap，因为该对象的线程安全应该由用户（程序员）自己来控制。

（2）Hashtable 的子类 Properties：

①作用：读写资源配置文件。

②键和值只能为字符串。

③方法：

setProperty(String key, String value)

getProperty(String key) //没有返回 null

getProperty(String key, String defaultValue) //没有返回 defaultValue

④读写资源配置文件方法：

.properties 文件

存储

store(OutputStream out, String comments) // comments 是备注

store(Writer writer, String comments)

读取

load(InputStream inStream)

load(Reader reader)

.xml 文件

存储

storeToXML(OutputStream os, String comment) //使用 UTF-8 字符集

storeToXML(OutputStream os, String comment, String encoding)

读取

LoadFromXML(InputStream in)

⑤示例：

```
//存储
Properties properties = new Properties();
properties.setProperty("name", "Tom");
properties.setProperty("age", "18");
//properties.store(new FileOutputStream(new File("info.properties")), "Tom's info");
properties.storeToXML(new FileOutputStream(new File("info.xml")), "Tom's info");

//读取
```

```
Properties properties = new Properties();
//properties.load(new FileInputStream(new File("info.properties")));
properties.loadFromXML(new FileInputStream(new File("info.xml")));
String name = properties.getProperty("name");
String age = properties.getProperty("age");
System.out.println("name:" + name + " age" + age);
```

⑥相对路径：
默认的当前路径位于当前工程的目录下，比如：
properties.store(new FileOutputStream(new File("info.properties")), "Tom's info");
会在工程目录下建立一个 **info.properties** 文件。

也可以使用类加载路径作为当前路径，比如：
properties.load(TestProperties.class.getResourceAsStream("info.properties"));
会在 TestProperties 类的字节码文件所在目录读取 **info.properties** 文件。

properties.load(TestProperties.class.getResourceAsStream("/info.properties"));
以及
properties.load(Thread.currentThread().getContextClassLoader().getResourceAsStream("info.properties"));
会在类所在的根路径（也就是最外层包所在目录中）读取 **info.properties** 文件。

17、排序

1）实体类自身可以排序

实体类想要实现排序功能，可以实现 Comparable 接口的 compareTo() 方法，这样，在排序方法中可以使用对象自带的 compareTo() 方法来排序。（也就是对象自己包含比较大小的方法）
A.compareTo(B)：
A<B，返回负数；
A=B，返回 0；
A>B，返回正数。

2）提供另外的比较器

可以在排序方法中传入一个实现了 Comparator 接口的 compare() 方法的比较器，然后在排序时使用比较器提供的 compare() 方法给带排序对象排序，而不是使用对象可能自带的 compareTo() 方法。（也就是使用比较器提供的比较对象大小的方法）
comparator. compare(A, B)：
A<B，返回负数；
A=B，返回 0；

A>B，返回正数。

3）Collections 工具类

提供了大量便于处理容器的方法，包括上述两种排序方法。
①public static <T extends Comparable<? super T>> void sort(List<T> list)
②public static <T> void sort(List<T> list, Comparator<? super T> c)

4）TreeSet 和 TreeMap

由于在 TreeSet 容器中添加元素时会自动给元素排序，所以要么元素实现了 Comparable 接口，自己就能够比较大小，要么在创建容器时给容器传一个比较器，使用该比较器来给元素比较大小，否则会由于无法排序而报错。TreeMap 容器要求 key 能够排序，和 TreeSet 同理。（注：**TreeSet 和 TreeMap 容器是在添加元素时给元素排序，一旦添加完成，若改变元素大小并不会改变原来的排序结果**）

18、对象的克隆

1）Object 类的 clone()方法

为了获取对象的一份拷贝，我们可以利用 Object 类的 clone() 方法。该方法是一个 protected native 方法，它直接在内存中进行二进制流的拷贝，不会执行构造函数，效率较高。

2）实体类如何实现克隆

首先，派生类实现 Cloneable 接口，这是一个标志性接口。然后，在派生类中覆盖基类的 clone() 方法（因为从其他包中通过继承得到的 protected 方法只能在本类中使用，无法再本包的其他类中调用），如果派生类的 clone() 方法需要在其他包中调用则声明为 public，在派生类的 clone()方法中，调用 super.clone()。

3）浅拷贝与深拷贝

Object 类的 clone() 方法是浅拷贝，只是将对象成员变量的值拷贝一份，包括引用成员变量，也只是拷贝一份引用。这样的话拷贝出来的对象和原始对象拥有同一个对象的引用，一般情况下并不符合我们的需求，所以为了实现深拷贝，应该在派生类的 clone() 方法中作出相应修改。注：String 类对象也是引用类型，但是它是不可变的，所以在拷贝对象中修改的话只会重新创建一个字符串，并不会影响原始对象，所以无需对该成员变量的拷贝作出相应修改。
示例代码：

```
//实现深拷贝
class Test
```

```
{
    public static void main(String[] args) throws CloneNotSupportedException
    {
        Teacher t = new Teacher("Tom");
        Student s1 = new Student("zhansan", 20, t);
        Student s2 = (Student) s1.clone();
        //修改拷贝对象的引用对象
        s2.teacher.name = "Jack";
        //输出原始对象的引用对象
        System.out.println(s1.teacher.name); //Tom  并未发生修改

    }
}

class Student implements Cloneable
{
    String name;
    int age;
    Teacher teacher;
    public Student(String name, int age, Teacher teacher)
    {
        super();
        this.name = name;
        this.age = age;
        this.teacher = teacher;
    }
    @Override
    protected Object clone() throws CloneNotSupportedException
    {
        Student s =    (Student) super.clone();
        s.teacher = (Teacher) s.teacher.clone();
        return s;
    }
}

class Teacher implements Cloneable
{
    String name;

    public Teacher(String name)
    {
        super();
        this.name = name;
    }
```

```
    @Override
    protected Object clone() throws CloneNotSupportedException
    {
        return super.clone();
    }
}
```

19、JNI（Java Native Interface）

1）native 方法不是在 Java 中编写的，而是用户可以在 Java 中调用的本地方法。
2）JNI 技术允许 Java 虚拟机内部运行的 Java 代码能够与其他编程语言编写的应用程序或库进行互操作，可以理解为 Java 和本地应用程序之间的中介。
3）写一个 native 方法的流程：
（1）写一个 Java 类，声明一个本地方法，并在静态块中导入库。

```
// HelloJNI.java
public class HelloJNI
{
    static
    {
        System.loadLibrary("hello"); //Load hello.dll (Windows) or libhello.so (Unixes)
    }

    private native void helloJNI(); //Declare a native method

    public static void main(String[] args)
    {
        new HelloJNI().helloJNI();    //Invoke the native method
    }
}
```

（2）编译代码。
命令：**javac HelloJNI.java**
（3）创建头文件。
命令：**javah -jni HelloJNI**
（4）根据生成的头文件，写一个本地方法的实现，并利用 visual studio 等 IDE 编译成 dll。

```
// HelloJNI.c
#include <jni.h>
#include <stdio.h>
#include "HelloJNI.h"

JNIEXPORT void JNICALL Java_HelloJNI_helloJNI(JNIEnv *env, jobject thisObj)
{
    printf("Hello JNI!\n");
```

```
    return;
}
```

（5）将生成的 hello.dll 与 Java 字节码文件放在同一目录下，运行测试。
命令：**java HelloJNI**

20、反射 reflection

1）Class 类

（1）在 Java 中，每个类（包括基本数据类型）都对应一个 Class 对象，该对象存储了这个类的类型信息。
（2）获取 Class 实例的三种方式：
① 利用对象调用 getClass() 方法获取该对象的 Class 实例；
② 使用 Class 类的静态方法 forName()，用类的名字获取一个 Class 实例；
③ 利用类名.class 的方式来获取 Class 实例，对于基本数据类型的封装类，还可以采用.TYPE 来获取相对应的基本数据类型的 Class 实例。
（3）在运行期间，如果我们要产生某个类的对象，JVM 会检查该类的 Class 对象是否已被加载。如果没有被加载，JVM 会根据类的名称找到.class 文件并加载它。一旦某个类型的 Class 对象已被加载到内存，就可以用它来产生该类型的对象。
（4）常用方法：
newInstance()：用来创建对象，会调用对应类的无参构造器；
newInstance(Object ... initargs)：传入参数，调用相应的构造器；

Constructor<?>[] getConstructors()：返回对应类的所有的 public 构造器；
Constructor<?>[] getDeclaredConstructors()：返回对应类的所有的构造器；

Method[] getMethods()：返回对应类的所有的 public 方法，包括从父类和父接口中继承而来的 public 方法；
Method[] getDeclaredMethods()：返回对应类的所有的方法，不包括从父类和父接口中继承而来的方法；
Method getMethod(String name, Class<?>... parameterTypes)：返回指定的 public 方法；
Method getDeclaredMethod(String name, Class<?>... parameterTypes)：返回指定方法；

Field[] getFields()：返回所有的 public 属性；
Field[] getDeclaredFields()：返回所有的属性；
Field getField(String name)：返回指定的 public 属性；
Field getDeclaredField(String name)：返回指定的属性；

boolean isPrimitive()：判断对应类是否为基本数据类型。

2）Constructor 类

常用方法：
Class<?>[] getParameterTypes()：返回构造器参数类型所对应的 Class 对象数组；
newInstance(Object ... initargs)：传入构造器的参数，创建一个实例对象。

3）Method 类

常用方法：
Object invoke(Object obj, Object... args)：传入方法的调用者以及参数，调用该方法，相当于：obj.method(args)；
setAccessible(boolean flag)：参数为 true 时，就算该方法私有的依然可以进行调用。

21、Runtime 类和 Process 类

1）每一个 Java 程序都有一个 Runtime 类的单一实例，它为程序提供了运行程序和运行环境之间的一个接口。
2）通过 Runtime.getRuntime() 来获取 Runtime 类的实例。
3）常用方法：
long freeMemory()：获取 JVM 的自由内存；
long totalMemory()：获取 JVM 的总内存；
long maxMemory()：获取 JVM 的最大内存；

一系列的 exec 方法：执行一个外部程序，并返回一个 Process 对象，用于管理子进程。

22、注解 Annotation

1）常用的三个内置注解

@Override:该注解只能用于方法，限定该方法只能为重写父类方法
@Deprecated: 用于表示某个程序元素（类, 方法等）已过时，不建议使用
@SuppressWarnings: 抑制编译器警告

2）元注解

（1）作用：用于注解其他注解。
（2）两个常用的元注解：
@Retention：用于描述注解的生命周期（即表示该注解保留到什么时候），可取值为：SOURCE（在源文件中保留，被编译器解析）、CLASS（在 class 文件中保留，被类加载器解析）、RUNTIME（在运行时保留，被程序解析，此时该注解可以被反射机制获取），这些值都定义在枚举类

RetentionPolicy 中。
@Target：用于描述注解的使用范围，可取值为：PACKAGE（包）、TYPE（类、接口、枚举）、FIELD（域）、METHOD（方法）、PARAMETER（参数）、CONSTRUCTOR（构造器）、LOCAL_VARIABLE（局部变量），这些值都定义在枚举类 ElementType 中。
（3）“@SuppressWarnings”注解的源码：

```
@Target({TYPE, FIELD, METHOD, PARAMETER, CONSTRUCTOR, LOCAL_VARIABLE})
@Retention(RetentionPolicy.SOURCE)
public @interface SuppressWarnings {
    String[] value();
}
```

注：其中 String[] value(); 表示的是在使用@SuppressWarnings 注解时所需的参数，String[] 表示参数类型，value 表示参数名。当注解的参数只有一个时，一般使用 value 作为参数名，因为在使用该注解时如果传入的参数不指定参数名的话默认为 value，如：
@Target({ElementType.METHOD, ElementType.FIELD})
等同于：
@Target(value={ElementType.METHOD, ElementType.FIELD})

3）自定义注解

使用@interface 关键词来自定义一个注解，默认实现了 java.lang.annotation.Annotation 接口，自定义时可以参考内置注解。
示例代码：

```
public class Test
{
    @MyAnnotation(b = 1, a = "aaa")
    int value;

    @MyAnnotation()
    void func()
    {
    }
}

@Target(value = { ElementType.METHOD, ElementType.FIELD })
@Retention(RetentionPolicy.RUNTIME)
@interface MyAnnotation
{
    String a() default "";
    int b() default 0;
}
```

4）注解的解析

如果只是定义了一个注解的话其实并没有用处，只有能够对该注解进行解析时注解才能够发挥作用。

使用反射获取注解信息的一个例子：

```
@MyAnnotation
public class Test
{
    @MyAnnotation(b = 1, a = "aaa")
    int value;

    public static void main(String[] args)
    {
        try
        {
            Class clazz = Class.forName("Test");
            //获取类的注解信息
            MyAnnotation ma = (MyAnnotation) clazz.getAnnotation(MyAnnotation.class);
            System.out.println(ma.a());
            System.out.println(ma.b());
            //获取属性的注解信息
            Field f = clazz.getDeclaredField("value");
            ma = f.getAnnotation(MyAnnotation.class);
            System.out.println(ma.a());
            System.out.println(ma.b());
        }
        catch (Exception e)
        {
            e.printStackTrace();
        }
    }
}

@Target(value = { ElementType.TYPE, ElementType.FIELD })
@Retention(RetentionPolicy.RUNTIME)
@interface MyAnnotation
{
    String a() default "";
    int b() default 0;
}
```

23、正则表达式

示例代码：

```
//利用正则表达式爬取网页的所有超链接
public class Test
{
    public static String getWebContent(String urlStr, String charset)
    {
        StringBuilder sb = new StringBuilder();
        try
        {
            URL url = new URL(urlStr);
            BufferedReader reader = new BufferedReader(new InputStreamReader(
                    url.openStream(), Charset.forName(charset)));
            String temp = "";
            while ((temp = reader.readLine()) != null)
            {
                sb.append(temp);
            }
        }
        catch (MalformedURLException e)
        {
            e.printStackTrace();
        }
        catch (IOException e)
        {
            e.printStackTrace();
        }
        return sb.toString();
    }

    public static List<String> getMatchResult(String destStr, String regexStr)
    {
        Pattern p = Pattern.compile(regexStr);
        Matcher m = p.matcher(destStr);
        List<String> result = new ArrayList<String>();
        while (m.find())
        {
            result.add(m.group(1));
        }
        return result;
    }
```

```
    public static void main(String[] args)
    {
        String destStr = getWebContent("http://www.163.com", "gbk");
        List<String> result = getMatchResult(destStr,
                "href=\"([\\w\\s./:]+?)\"");
        for (String temp : result)
        {
            System.out.println(temp);
        }
    }
}
```