JAVA考试

- 题型（五个判断（2‘），编程（6个小题）30分，15个单选（一个2分），问答题（4/5 侧重基础概念），五个填空（2’））
- 常量变量引用的基本概念
  - 常量赋值的方法
- 数据类型
  - 基本型
    - 数值型
      - 整数类型（byte,short,int,long)
      - 浮点类型（float,double)
        - 如果表示float类型，则需要在字面值末尾加上大小写的F
        - double的范围比float大，在赋值时，double不能赋值给float，但float的数据可以赋值给double
    - 字符型（char)
    - 布尔型（boolean)
      - 只有true和false
  - 引用类型
    - 类（class)

```
public class MyClass {
    // 类的成员变量和方法
}
```

    - 接口（interface)

```
public interface MyInterface {
    // 接口的方法
}
```

    - 数组（[])

```
int[] myArray = new int[10];
```

  - 数组是什么
    - 数组是一种引用类型。元素按线性顺序排序。所谓线性就是除第一个元素外，每个元素都有唯一的前驱元素；除最后一个元素外，每一个元素都有唯一的后继元素（一个跟一个）

- 初始化
  - int[] arr = new int[3]; // 0,0,0
  - int[][] arr2 = new int[][]{{1,2,3},{4,5,6}};
- 冒泡排序

```
        }
        for(int i=0;i<arr.length-1;i++) { //控制冒泡轮数
            for(int j=0;j<arr.length-1-i;j++) { //控制每轮的次数
                if(arr[j]>arr[j+1]) { //每次都是和它的下一个元素比
                    int t = arr[j]; //符合条件则交换
                    arr[j] = arr[j+1];
                    arr[j+1] = t;
                }
            }
        }
        System.out.println("排序后：");
        for(int i=0;i<arr.length;i++) {
            System.out.println(arr[i]); //输出排序后的数组
        }
```

- void类型？算不算数组？是不是类型？
  - Void是一个类，是一个不能实例化的占位符类，它持有对一个类对象的引用，这个类对象代表java关键字void。
  - 如果函数返回值是Void，那么函数必须明确返回null。
- 表达式 运算符 优先级
  - 算术运算符
    - + - * / %
    - 注意
      - 都是二元运算符，使用时必须要有左右两个操作数
      - int / int 结果还是int类型，而且会向下取整
      - 做除法和取模时，右操作数不能为0
  - 增量运算符
    - += -= *= %=
    - 只有变量才能使用该运算符，常量不能使用
  - 关系运算符：
    - == 、 ！= 、 > 、 < 、 <= 、 >=
    - 注意：关系运算符的结果为布尔值类型
  - 自增自减运算符
    - ++ - -
    - 注意
      - 如果单独使用， 【前置++】和【后置++】没有任何区别
      - 如果混合使用， 【前置++】先+1，然后使用变量+1之后的值（**++用于变量前时，应该先+1，再进行其他运算**）

        - 【后置++】先使用变量原来的值，表达式 结束时给变量+1（**++用于变量后时，应该先进行其他运算，再+1**）
    - 逻辑运算符
        - && || !
        - 运算结果都是 boolean类型
    - 条件运算符
        - 表达式1 ? 表达式2 : 表达式3
        - 当 表达式1 的值为 true 时, 整个表达式的值为 表达式2 的值;
        - 当 表达式1 的值为 false 时, 整个表达式的值为 表达式3 的值.
    - 优先级
        - **括号级别最高，逗号级别最低，单目 > 算术 > 位移 > 关系 > 逻辑 > 三目 > 赋值**

优先级	运算符	结合性
1	() [] .	从左到右
2	! ~ ++ --	从右到左
3	* / %	从左到右
4	+ -	从左到右
5	<< >> >>>	从左到右
6	< <= > >= instanceof	从左到右
7	== !=	从左到右
8	&	从左到右
9	^	从左到右
10	|	从左到右
11	&&	从左到右
12	||	从左到右
13	? :	从左到右
14	= += -= *= /= %= &= |= ^= ~= <<= >>= >>>=	从右到左
15	,	从右到左

- 类型转换（例如 double和int间转换）
    - 隐式类型转换
        - 如果将一个小类型的变量赋值给一个大类型的变量，Java 会自动进行类型转换，这种转换叫做隐式类型转换。

```
int num = 10;
double d = num;
```

        - 如果低级类型为char型，向高级类型(整型)转换时，会转换为对应ASCII码值

```
char c='c';
int i=c;
System.out.println("output: " i);//output: 99
```

    - 显示类型转换

- 如果将一个大类型的变量赋值给一个小类型的变量，Java 不会自动进行类型转换，必须使用显式类型转换。

```
double d = 10.5;
int num = (int) d;
```

- 目标类型 变量=（目标类型）值
- 需要使用强制类型转换运算符将 d 转换成 int 类型
- 不同类型变量混合后得到的结果是**精度最高**的类型

- **包装类过度类型转换**
  - 在 Java 中，有些数据类型是基本数据类型，例如 int、double 等，而有些数据类型是对象类型，**例如 Integer、Double 等。这些对象类型称为包装类，**它们是为了让基本数据类型具有对象特性而设计的
  - 优点
    - 包装类可以将基本数据类型转换成对象类型，从而可以将基本数据类型作为对象来处理
    - 包装类提供了一些方法，**例如 toString()、valueOf()** 等，可以方便地对基本数据类型进行操作
    - 包装类可以作为泛型参数，从而使泛型可以处理基本数据类型
  - 包装类

基本数据类型	包装类
boolean	Boolean
byte	Byte
char	Character
short	Short
int	Integer
long	Long
float	Float
double	Double

    - 基本数据类型转换成包装类

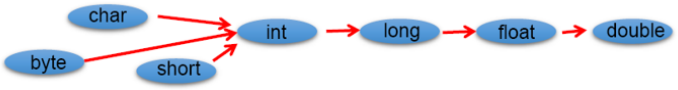
```
int num = 10;
Integer i = Integer.valueOf(num);
```

      - String strAge = String.valueof(age)
    - 包装类转换成基本数据类型

```
Integer i = 10;
int num = i.intValue();
```

        - int age = Integer.parseInt(strAge)
        - float money = Float.parseFloat(strMoney)
    - 强制转换
        - **强制类型转换**是A类型的数据表示范围比B类型大，则将A类型的值赋值给B类型，则需要强制类型转换。
            - 低精度的值可以直接赋给高精度变量，直接转换为高精度

            byte < short < char < int < long < float < double

        - short类型和char类型互相转化需要强制转换
        - 基本类型和引用类型不可以互相转换
- new关键字
    - 创建一个Java对象需要三部：声明引用变量、实例化、初始化对象实例
    - 使用new + 构造函数 来创建一个对象

```
public class Cat {

        public Cat() {

                System.out.println("这是构造方法");

        }

```

- 构造函数
    - 构造函数的名字必须与定义他的类名完全相同，没有返回类型，甚至连void也没有
    - 构造函数的调用是在创建一个对象时使用new操作进行的。构造方法的作用是初始化对象

      Person p =new Person(); person()调用的就是Person的构造方法

    - 不能被static、final、synchronized、abstract和native修饰。构造函数不能被子类继承
    - 构造函数在创建对象时自动执行,一般不能显式地直接调用.。 构造方法可以被重载。没有参数的构造方法称为默认构造方法

Java构造函数	方法
构造函数用于初始化对象的状态。	方法用于暴露对象的行为。
构造函数不能有返回类型。	方法必须有返回类型。
构造函数被隐式调用。	该方法被显式调用。
如果类中没有任何构造函数，Java 编译器会提供默认构造函数。	在任何情况下，编译器都不会提供该方法。
构造函数名必须与类名相同。	方法名可能与类名相同，也可能不同。

- 创建对象调用构造函数

- 类型
  - 默认构造函数（无参数构造函数）
  - 参数化构造函数

```
//Java程序演示参数化构造函数的使用。
class Student4{
    int id;
    String name;
    //创建参数化构造函数
    Student4(int i,String n){
    id = i;
    name = n;
    }
    //显示值的方法
    void display(){System.out.println(id+" "+name);}

    public static void main(String args[]){
    //创建对象并传递值
    Student4 s1 = new Student4(111,"小卡");
    Student4 s2 = new Student4(222,"小安");
    //调用方法显示对象的值
    s1.display();
    s2.display();
   }
}
```

- 语法
  - 循环分支，选择
  - catch-finally异常捕获语句
    - 格式 **无论是否发生异常，finally 代码块中的代码总会被执行**

```
try {
    // 可能抛出异常的代码
} catch (ExceptionType1 e1) {
    // 处理异常类型1
} catch (ExceptionType2 e2) {
    // 处理异常类型2
} finally {
    // 可选的finally块，用于执行无论是否发生异常都需要执行的代码
}
```

```
try {
    System.out.println("请输入学生姓名：");
    name = scanner.next();
    System.out.println("请输入学生年龄：");
    age = scanner.nextInt();
    System.out.println("请输入学生性别：");
    sex = scanner.next();
} catch (Exception e) {
    e.printStackTrace();
    System.out.println("输入有误！");
}
System.out.println("姓名：" + name);
System.out.println("年龄：" + age);
```

        - catch中的异常类型如果满足子父类关系，则要求子类一定要声明在父类的上面，否则，报错
    - 循环三种的基本形式
    - Switch

```
switch (表达式) {
        case 常量表达式或枚举常量:
            语句;
            break;
        case 常量表达式或枚举常量:
            语句;
            break;
        ......
        default: 语句;
            break;
    }
```

        - **一直执行到break语句处或者是switch语句的末尾**，如果表达式的值与所有的case标签的值都不匹配，（如果存在default语句的情况），则执行default语句下面的代码语句块
        - default分支语句原则上可以写switch分支中的任意位置，如果没有写到最后一个 case分支语句 的下面的话！后面就必须要加上break关键字
- 各种修饰符
    - abstract
        - 用处
            - **不想类能够实例化时**(类实例化没有意义,例如Animal类),就在class前面加上abstract关键字
            - **不想方法能够被对象.方式调用**并且要求能**实例化的子类一定要重写该方法**时,就把abtract写在方法名的前面(抽象方法所在的类一定是抽象类)
        - 修饰类

```
权限修饰符 abstract class 类名{
}
```

        - 修饰方法

```
权限修饰符 abstract 方法返回值类型 方法名(形参列表);
//抽象方法是没有方法体的
//能实例化对象的类继承抽象类时,会重写抽象类的所有抽象方法
```

        - 抽象方法所在的类一定是抽象类,抽象类可以一个抽象方法都没有
            - 抽象方法是没有方法体的方法,是不能通过对象名.方法的方式去调用的,而若所在的类不是抽象类,意味着它可以实例化对象,实例化对象后就可以通过对象名.方法名来调用

方法,这**岂不是自相矛盾**吗?抽象类只是让该类不能实例化对象了而已,里面的抽象方法有还是没有是不会有影响的

- 抽象方法一定要求被非抽象的子类重写,抽象子类可以不重写
  - 抽象方法重写的目的是规范子类,是你一定要去具体实现的,若一直都是抽象子类继承的话,那这样设计就没有任何意义了。其实,抽象方法会强制非抽象的子类去重写该方法,不重写的话编译就通不过。
- **抽象类可以继承抽象类或者非抽象类吗?**
  - 把抽象类当成一个特殊的类就行,特殊在于该类不能实例化对象,但是继承什么的非抽象的规则是一样的,你不显示的继承父类时,它默认继承于java.lang.Object
- **抽象方法可以在抽象类中重载吗?**
  - 可以的,抽象方法可以看成一个特殊的方法,特殊在它要求继承它的非抽象子类一定要重写实现该方法,至于其他的部分其实和普通方法是一样的用法,因此也可以实现方法的重载。
- 抽象类一定有构造方法,这是为了子类而服务的
  - 因为子类继承抽象父类后,**调用自己的构造方法前,会默认调用父类构造方法**,模拟的是先有父后又子,而调用的父类构造方法是给子类的父类型特征区里面继承过来的属性进行初始化赋值。因此可以说是为了子类而服务的

- final（final是不能作为父类的）
  - final可以用来修饰的结构：类、方法、变量
    - final用来修饰一个类：此类不能被其它类继承。
    - final 用来修饰方法 ： 表明此方法不可以被重写
    - final 用来修饰变量 ， 此时变量就相当于常量

```
class A {
    public final void f1() {
    }
}
public class B extends A {
    public void f1() { // 编译出错，因为f1不可以被重写
    }
    public final void f1() {
    }
    public void f2() {
        f1(); // 编译通过，final会被继承给子类，子类可以直接调用。
    }
}
```

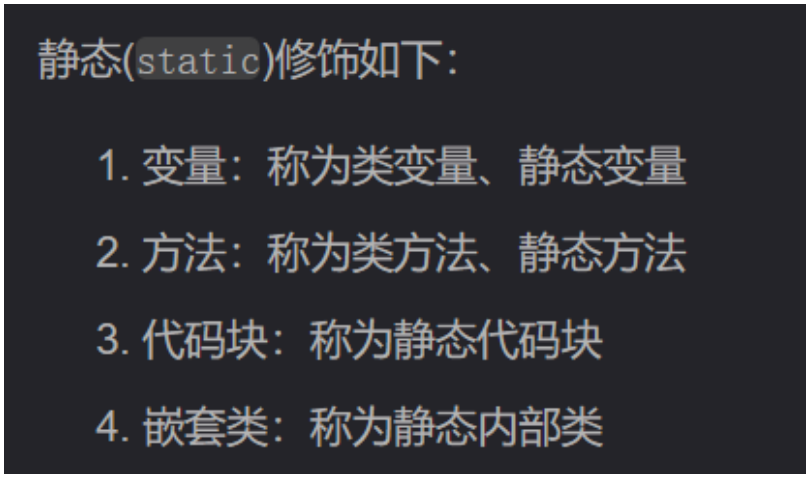
  - 但要注意：final类中所有的成员方法都会隐式的定义为final方法。比如：**String类、System类、StringBuffer类**
- finally
  - 一定会被执行的代码
    - 即使catch 中又出现异常了， try中有return语句， catch中有return语句等情况
  - 用于输入输出流
- string不能被继承

- String类是不可被继承的。原因在于Java中的String类是final类。final阻止了任何继承行为的发生

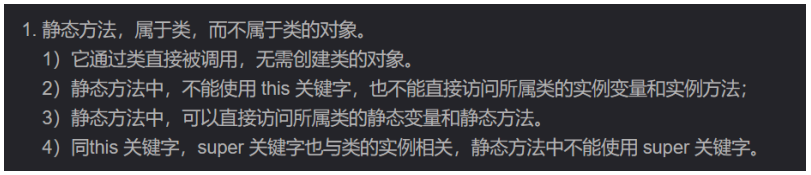
- 访问叔限
  - public > protected> default(包访问权限) > private

访问权限	含义	本类	本包的类	非本包子类	非本包非子类
public	公共的	是	是	是	是
protected	保护访问权限	是	是	是	否
default	包访问权限	是	是	否	否
private	私有的	是	否	否	否

  - 如果省略了访问修饰符，那访问权限就是default
- static
  - 静态方法 不用定义对象类名调用 可不抽象



    - 静态变量（节省内存）
      - 在类的内部，可以在任何方法内直接访问静态变量
      - 在其他类中，可以通过类名访问该类中的静态变量
    - 静态方法（不能使用this,super)



  - 成员方法 创建对象再调用
    - 成员方法需要先new再使用？调用方法
      - 成员方法是属于类的实例的，而不是属于类本身的。要调用成员方法，通常需要先创建类的实例（对象），然后通过该对象来调用方法。

```java
public class MyClass {
    // 成员变量
    private int myVariable;

    // 成员方法
    public void myMethod() {
        System.out.println("调用了成员方法");
    }

    public static void main(String[] args) {
        // 创建类的实例
        MyClass myObject = new MyClass();

        // 调用成员方法
        myObject.myMethod();
    }
}
```

- 非static方法？(实例方法)

- 类的定义 概念
  - 继承
    - 继承就是子类继承父类的特征和行为，使得子类对象（实例）具有父类的实例域和方法，或子类从父类继承方法，使得子类具有父类相同的行为

```
class 父类{
    ...       //成员变量、成员方法
}
class 子类 extends 父类{
    ...       //类体
}
```

    - 超类
      - 关键字extends表明正在构造的新类派生于一个已存在的类。这个类称为超类、基类或父类
    - 子类，派生类
      - 新类称为子类、派生类或孩子类。
    - 子类调用父类的方法需要使用关键字super， **使用super调用构造器必须是子类构造器的第一条语句**
  - 派生
    - 子类的对象构造顺序
      - 执行顺序：先执行父类构造方法，再执行子类构造方法。在多层继承层时，编译器会一直上溯到最初类，再从“上”到“下”依次执行。

- 子类的继承性
  - **若子类和父类在同一个包内，子类可以继承父类中访问权限设定为public、protected、 default的成员变量和方法**
  - **若子类和父类不在同一个包内，子类可以继承父类中访问权限设定为public、protected的成员变量和方法**
- 成员变量的隐藏和方法的重写
  - 方法的重写
    - **重写和重载的区别：重载是在本类之中，重写是在父类和子类之间**
    - 方法重写是指：子类中定义一个方法，并且这个方法的名字、放回类型、参数个数和类型与从父类中继承的方法完全相同
    - 如果子类想使用被隐藏的方法就必须使用关键字super
    - 要求

      > 方法重写的一些要求：两同两小一大：
      > **两同**：方法名相同，参数列表一致。
      > **两小**：子类返回值类型更小或相等，子类抛出异常小于等于父类的抛出异常
      > **一大**：子类的访问权限比父类大或者相等。
  - 成员变量的隐藏
    - 变量只能被隐藏，不能被重写
    - 静态方法只能被隐藏不可被重写
    - 不能用子类的静态方法隐藏父类的非静态方法
    - 不能用子类的非静态方法覆盖父类的静态方法
- 多态
  - 多态的前提条件：1、子类继承父类　2、子类重写父类方法、　3、父类引用指向子类对象多态的
  - 重载overload
    - 重载是在**同一个类里同名的方法，但是要形参数量不一致**
  - 覆盖（重写）overwrite
    - 重写就是把子类有一个方法，和父类的某个**方法名称，返回类型，参数一样**
    - 子类覆盖父类 子类覆盖接口

```java
class Animal {
    void makeSound() {
        System.out.println("Animal makes a sound");
    }
}

class Dog extends Animal {
    // 覆盖父类的方法
    @Override
    void makeSound() {
        System.out.println("Dog barks");
    }
}

public class Main {
    public static void main(String[] args) {
        Animal animal1 = new Animal();
        Animal animal2 = new Dog();

        // 调用被覆盖的方法，动态绑定确定调用哪个版本
        animal1.makeSound(); // 输出: Animal makes a soun
        animal2.makeSound(); // 输出: Dog barks
    }
}
```

- 同一个信息，不同的响应，用多态实现

- 类类型转换
  - 向上转型
    - 父类的引用指向子类的对象，把一个子类对象直接赋给一个父类引用变量，无须任何类型转换（隐式的）

```
父类类型 引用名 = new 子类类型();
//右侧创建一个子类对象，把它当作父类看待使用
```

```java
class Animal {
    public void eat() {
        System.out.println("Animal is eating");
    }
}

class Dog extends Animal {
    public void bark() {
        System.out.println("Dog is barking");
    }
}
```

```java
Dog dog = new Dog();
Animal animal = dog; // 上转型
animal.eat(); // 输出 "Animal is eating"
```

```java
// 定义一个父类
class Animal {
    // ...
}

// 定义一个子类
class Dog extends Animal {
    // ...
}

public class Main {
    public static void main(String[] args) {
        // 向上转型
        Animal animal = new Dog(); // 子类对象赋给父类类型的
    }
}
```

- 可以调用父类的所有成员（需遵守访问权限）
- **不能调用子类的特有成员**

- 向下转型
  - 一个已经向上转型的子类对象，将**父类引用转为子类引用**，试图把一个父类实例转换成子类类型，则这个对象对象必须实际上是子类实例才行（即编译时类型为父类类型，而运行时类型是子类类型），**否则将在运行时引发ClassCastException（强制类型转换错误）异常**

```
子类类型 引用名 = （子类类型）父类引用;
//用强制类型转换的格式，将父类引用类型转为子类引用类型
```

```java
class Animal {
    public void eat() {
        System.out.println("Animal is eating");
    }
}

class Dog extends Animal {
    public void bark() {
        System.out.println("Dog is barking");
    }
}
```

```java
Animal animal = new Dog();
if (animal instanceof Dog) {
    Dog dog = (Dog) animal; // 下转型
    dog.bark(); // 输出 "Dog is barking"
}
```

```java
// 向下转型的例子
public class Main {
    public static void main(String[] args) {
        Animal animal = new Dog();

        // 向下转型
        if (animal instanceof Dog) {
            Dog dog = (Dog) animal; // 强制类型转换
            // 现在可以使用 dog 对象了
        } else {
            System.out.println("对象不是 Dog 类型");
        }
    }
}
```

- 只能**强制转换父类的引用**，不能强制转换父类的对象
- 要求父类的引用**必须指向**的是**当前目标类型的对象**
- 当向下转型后，**可以调用子类**类型中**所有的成员**
- instanceof 比较操作符
  - 为了**避免**上述**类型转换异常**的问题，我们引出 **instanceof** 比较操作符，用于**判断对象的类型是否为XX类型或XX类型的子类型**

- 父类型 子对象

当Java虚拟机（JVM）加载并初始化一个类及其父类时，它会按照以下顺序执行：

1. 父类的静态变量和静态代码块按照在代码中出现的顺序依次执行。
2. 子类的静态变量和静态代码块按照在代码中出现的顺序依次执行。
3. 父类的成员变量和非静态代码块按照在代码中出现的顺序依次执行。
4. 父类的构造函数执行。
5. 子类的成员变量和非静态代码块按照在代码中出现的顺序依次执行。
6. 子类的构造函数执行。

- 接口
  - 接口是啥
    - 接口可以理解为一种特殊的类，里面全部是由**全局常量**和**公共的抽象方法**所组成。接口是解决**Java无法使用多继承**的一种手段
    - 就像一个类一样，一个接口也能够拥有方法和属性，但是在接口中声明的方法默认是抽象的。（**即只有方法标识符，而没有方法体**）
  - 接口允不允许一个方法都没有

- 允许，标记接口MarkerInterface 允许一个方法都没有表明从属关系 仅用于标识类是否属于某个特定的类别

```
// 标记接口
interface MarkerInterface {
    // 这里没有任何方法
}

// 实现标记接口的类
class MyClass implements MarkerInterface {
    // 实现类的具体逻辑
}
```

- 实现接口中的抽象方法
  - **接口是百分之百的抽象类**
  - 如果一个类实现了一个接口，但没有提供接口中某个抽象方法的实现，那么该类必须声明为抽象类。抽象类本身不需要提供接口中的所有方法的实现，但它可以留给子类来实现。

```
// 定义一个接口
interface MyInterface {
    void abstractMethod();
}

// 实现接口的类
class MyClass implements MyInterface {
    // 实现接口中的抽象方法
    @Override
    public void abstractMethod() {
        System.out.println("Implementation of abstractM
    }

    // 这里可以添加类自己的其他方法和属性
}

public class Main {
    public static void main(String[] args) {
        // 创建实现接口的类的实例
        MyClass myObject = new MyClass();

        // 调用实现的方法
        myObject.abstractMethod(); // 输出: Implementati
    }
}
```

- 接口中允许非静态方法
  - 默认方法

```java
// 接口中的默认方法
interface MyInterface {
    void abstractMethod();

    default void defaultMethod() {
        System.out.println("Default implementation of def
    }
}

// 实现接口的类
class MyClass implements MyInterface {
    // 提供抽象方法的具体实现
    @Override
    public void abstractMethod() {
        System.out.println("Implementation of abstractMet
    }
}
```

    - 不考函数式接口，拉姆达表达式
- 类的构造函数
    - new一个对象调用构造函数
    - 规划
        - 需不需要显式定义
            - 如果你为类显式定义了构造函数，Java 将不再提供默认构造函数。

```java
class MyClass {
    // 显式定义的构造函数
    public MyClass() {
        // 构造函数的初始化逻辑
    }

    // 也可以提供带参数的构造函数
    public MyClass(int value) {
        // 构造函数的初始化逻辑
    }
}
```

        - 有超类要调用超类构造函数
            - 第一行必须是

```java
class SuperClass {
    public SuperClass(int value) {
        // 超类的构造函数
    }
}

class SubClass extends SuperClass {
    public SubClass(int value) {
        super(value); // 调用超类的构造函数
        // 子类的构造函数的其他初始化逻辑
    }
}
```

- 构造函数调用规则 超类构造函数 必须在第一行调用etc 调用规则

- 内部类
  - **匿名类**
    - 本质上是一个没有名字的局部内部类，定义在方法中、代码块中、等。

```
new 类|抽象类名|或者接口名（）{
重写方法;
}
```

```
Employee a = new Employee(){
public void work(){
}
}
```

    - 匿名内部类可以作为方法的实际参数进行传输。
  - **静态内部类**
    - 它的特点和使用与普通类是完全一样的

```
public class Outer{
//静态内部类
public static class Inner{
}
}
```

2、静态内部类创建对象的格式：

格式：外部类名.内部类名 对象名 = new 外部类.内部类构造器;
范例： Outer.Inner in = new Outer.Inner();

- 静态内部类中是否可以直接访问外部类的静态成员?
  - 可以，外部类的静态成员只有一份，可以被共享访问
- 静态内部类中是否可以直接访问外部类的实例成员?
  - 不可以，外部类的实例成员必须要用外部类对象访问

- 成员内部类
  - 无 static 修饰，属于外部类的对象

```
public class Outer{
//成员内部类
public class Inner{
}
}
```

格式：外部类名.内部类名 对象名 = new 外部类构造器.new 内部类构造器（）;
范例：Outer.Inner in = new Outer().new.Inner()

  - 成员内部类中是否可以直接访问外部类的静态成员?
    - 可以，外部类的静态成员只有一份，可以被共享访问。
  - 成员内部类的实例方法是否可以直接访问外部类的实例方法?
    - 可以，因为必须现有外部类对象，才能有成员内部类对象，所以可以直接访问外部类对象的实例成员。

- 抽象接口
  - 抽象接口 接口方法可非抽象，静态方法
- 静态非静态方法
  - 静态方法不能访问非静态成员（非静态变量或方法），因为它们没有隶属于特定的对象实例

```
class MyClass {
    static void staticMethod() {
        System.out.println("This is a static method");
    }
}

public class Main {
    public static void main(String[] args) {
        MyClass.staticMethod(); // 调用静态方法
    }
}
```

- 非静态方法可以访问类的非静态成员和静态成员

```
class MyClass {
    void nonStaticMethod() {
        System.out.println("This is a non-static method"
    }
}

public class Main {
    public static void main(String[] args) {
        MyClass myObject = new MyClass();
        myObject.nonStaticMethod(); // 调用非静态方法
    }
}
```

- 多线程
  - 基本概念
    - 进程
      - 进程就是正在运行中的程序（进程是驻留在内存中的）
    - 线程
      - 线程就是进程中的单个顺序控制流，也可以理解成是一条执行路径
    - 单线程：一个进程中包含一个顺序控制流（一条执行路径）
    - 多线程：一个进程中包含多个顺序控制流（多条执行路径）
  - 构造方法
    - **无参构造方法**

```
Thread()
```

    - `带有Runnable参数的构造方法`

```
Thread(Runnable target)
```

```
Runnable myRunnable = /* 实现 Runnable 接口的对象 */;
Thread myThread = new Thread(myRunnable);
myThread.start();
```

- 带有线程名字的构造方法

```
Thread(String name)
```

```
Thread myThread = new Thread("MyThread");
```

- 带有Runnable和线程名字的构造方法

```
Thread(Runnable target, String name)
```

```
Runnable myRunnable = /* 实现 Runnable 接口的对象 */;
Thread myThread = new Thread(myRunnable, "MyThread");
```

- 带有ThreadGroup、Runnable和线程名字的构造方法：

```
Thread(ThreadGroup group, Runnable target, String name)
```

```
ThreadGroup myGroup = new ThreadGroup("MyGroup");
Runnable myRunnable = /* 实现 Runnable 接口的对象 */;
Thread myThread = new Thread(myGroup, myRunnable, "MyTh
```

- 启动方法
  - start()方法。单纯调用run()方法不会启动线程，不会分配新的分支栈
- 构造一个线程的基本方式
  - Thread类

```
class MyThread extends Thread {
    public void run() {
        // 线程执行的任务
    }
}

// 创建线程并启动
MyThread myThread = new MyThread();
myThread.start();
```

  - Runnable接口

```java
public class Demo02 {
    public static void main(String[] args) {
        MyRunnable myRun = new MyRunnable();//将一个任务提取出来，让多个线程共同去执行
        //封装线程对象
        Thread t01 = new Thread(myRun, "线程01");
        Thread t02 = new Thread(myRun, "线程02");
        Thread t03 = new Thread(myRun, "线程03");
        //开启线程
        t01.start();
        t02.start();
        t03.start();
        //通过匿名内部类的方式创建线程
        new Thread(new Runnable() {
            @Override
            public void run() {
                for (int i = 0; i < 20; i++) {
                    System.out.println(Thread.currentThread().getName() + " - " + i);
                }
            }
        },"线程04").start();
    }
}
//自定义线程类，实现Runnable接口
//这并不是一个线程类，是一个可运行的类，它还不是一个线程。
class MyRunnable implements Runnable{
    @Override
    public void run() {
        for (int i = 0; i < 50; i++) {
            System.out.println(Thread.currentThread().getName() + " - " + i);
        }
    }
}
```

```java
class MyRunnable implements Runnable {
    public void run() {
        // 线程执行的任务
    }
}

// 创建线程并启动
Thread myThread = new Thread(new MyRunnable());
myThread.start();
```

- **Callable和Future接口**

```java
import java.util.concurrent.Callable;
import java.util.concurrent.FutureTask;
```

```
import java.util.concurrent.Callable;
import java.util.concurrent.Future;
import java.util.concurrent.FutureTask;

class MyCallable implements Callable<String> {
    public String call() throws Exception {
        // 线程执行的任务，可以返回结果
        return "Task completed";
    }
}

// 创建线程并启动
FutureTask<String> futureTask = new FutureTask<>(new MyC
Thread myThread = new Thread(futureTask);
myThread.start();

// 获取线程执行结果
String result = futureTask.get();
```

- 这种方式的优点：可以获取到线程的执行结果
- 这种方式的缺点：效率比较低，在获取t线程执行结果的时候，当前线程受阻塞，效率较低。

```
        FutureTask task = new FutureTask(new Callable() {
            @Override
            public Object call() throws Exception { // call()方法就相当于run方法。只不过
                // 线程执行一个任务，执行之后可能会有一个执行结果
                // 模拟执行
                System.out.println("call method begin");
                Thread.sleep(1000 * 10);
                System.out.println("call method end!");
                int a = 100;
                int b = 200;
                return a + b; //自动装箱(300结果变成Integer)
            }
        });

        // 创建线程对象
        Thread t = new Thread(task);

        // 启动线程
        t.start();

        // 这里是main方法，这是在主线程中。
        // 在主线程中，怎么获取t线程的返回结果？
        // get()方法的执行会导致“当前线程阻塞”
        Object obj = task.get();
        System.out.println("线程执行结果:" + obj);
        // main方法这里的程序要想执行必须等待get()方法的结束
        // 而get()方法可能需要很久。因为get()方法是为了拿另一个线程的执行结果
        // 另一个线程执行是需要时间的。
        System.out.println("hello world!");
    }
```

- 线程的几种状态 状态转换
  - **新建状态（New）：**
    - 当线程对象被创建但还没有调用start()方法时，线程处于新建状态。
    - 可以通过创建Thread类的实例或者通过实现Runnable接口并创建Thread实例来得到新建状态的线程。

        - **就绪状态（Runnable）：**
            - 当线程对象调用了start()方法后，线程处于就绪状态。
            - 处于就绪状态的线程可能在任何时刻被调度执行，但具体的执行时间由操作系统的线程调度器决定。
        - **运行状态（Running）：**
            - 当线程获得CPU时间片并开始执行时，线程处于运行状态。
            - 处于运行状态的线程执行自己的任务。
        - **阻塞状态（Blocked）：**
            - 线程在执行过程中，因为某些原因放弃CPU使用权，暂时停止执行，进入阻塞状态。
            - 可能由于等待某个资源、等待用户输入、等待某个条件满足等原因。
        - **等待状态（Waiting）：**
            - 线程因为调用wait()方法而进入等待状态。
            - 可以通过notify()或notifyAll()方法唤醒等待状态的线程。
        - **超时等待状态（Timed Waiting）：**
            - 线程因为调用了具有超时参数的sleep()、join()、wait()方法而进入超时等待状态。
            - 在指定的时间后，线程会自动转换为就绪状态。
        - **终止状态（Terminated）：**
            - 线程执行完任务或者发生了未捕获的异常导致线程终止。
            - 线程完成执行或者发生异常后进入终止状态，不再运行
    - 线程的调用方式
        - 处理机主动调用（主动配合运行时 主动让其处理）
        - 主动让出处理机（等待处理机）（ 优先级队列分时调度）
    - 线程同步 线程池不考
- IO流基本概念（会考相关内容）
    - 如何分类
        - 按数据流的方向：输入流、输出流
            - **读取文件是输入流，写文件是输出流**
        - 按处理数据单位：字节流、字符流
        - 按功能：节点流、处理流
            - **节点流**：直接操作数据读写的流类，比如FileInputStream
            - **处理流**：对一个已存在的流的链接和封装，通过对数据进行处理为程序提供功能强大、灵活的读写功能，例如BufferedInputStream（缓冲字节流）
                - 普通流每次读写一个字节，而缓冲流在内存中设置一个缓存区，缓冲区先存储足够的待操作数据后，再与内存或磁盘进行交互。这样，在总数据量不变的情况下，通过提高每次交互的数据量，减少了交互次数。
    - 如何使用

    - **FileInputStream、FileOutputStream（字节流）BufferedInputStream、BufferedOutputStream（缓冲字节流）**
  - try捕获 凡是io流都要这样，必须异常捕获
  - 基本io流
  - inputstream
  - outputsteram
- 字符串
  - string 不可变
  - stringbuffer 可变
  - 文件及字符串处理 编程题
    - 统计单词个数
    - 结合io流 文件打开与关闭 如何读字符串 如何写入?
  - 中缀表达式变后缀表达式
  - 文件查重