java笔记

java的多态性质

- 一个对象的编译类型与运行类型可以不一致。
- 编译类型在定义对象时，就确定了，不能改变，而运行类型是可以变化的，编译类型看左边，运行类型看右边。
- 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（包括属性和方法）**，运行效果看子类的具体实现。
- 一个已经向上转型的子类对象，将**父类引用转为子类引用**是向下转型。注意，只能**强制转换父类的引用**，不能强制转换父类的对象，因此要求父类的引用**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。**

举个例子：

```
//转型演示
public class Test02 {
    public static void main(String[] args) {
        //向上转型（自动类型转换）
        Person p1 = new Student();

        //调用的是 Student 的 mission
        p1.mission();

        //向下转型
        Student s1 = (Student)p1;

        //调用的是 Student 的 mission
        (Person) s1.mission();

        //调用的是 Student 的 score （Student独有）
        s1.score();
    }
}
```

- 一个多态的易错例子：

```
public class Test7 {
    public static void main(String[] args){
        new B().display();
    }
}
class A{
    public void draw() {
        System.out.print("Draw A.");
    }
    public void display() {
        draw();
        System.out.print("Display A.");
    }
}
class B extends A{
```

```
    public void draw() {
        System.out.print("Draw B.");
    }
    public void display() {
        super.display();
        System.out.print("Display B.");
    }
    }
// 程序输出结果是：Draw B.Display A.Display B.          原因分析：main方法中创建了类B的一个实例，并调用了该实例的 display方法。类B的display方法首先调用 super.display()，即调用父类A的display方法。父类A的display方法首先调用 draw()方法。由于这是**在B的实例上下文中调用**的，所以会调用B类的draw方法，输出“Draw B.”。父类A的display方法接着输出 “Display A.”。最后，返回到类B的display方法，并输出 “Display B.”。
```

java的属性和方法访问权限

访问权限	本类	本包	子类	它包
public	√	√	√	√
protected	√	√	√	X
包级(默认)	√	√	X	X
private	√	X	X	X

- main函数不一定要在public类中

java对象和类

- this不能调用静态方法，静态方法不具有多态性。
  举个例子：

```
public class Test_Hide_Override {
    public static void main(String... args){
        A o = new C();
        o.m1();          //①    由于声明是A类型，所以调用的是A的m1
        o.m2();          //②    由于多态性质，调用的是C的m2
        ((B)o).m1();         //③   由于声明类型变为B，调用的是B的m1
        ((A)(B)o).m1();      //④    由于声明类型变为A，调用的是A的m1
        ((A)(B)o).m2();      //⑤     由于多态性质，调用的是C的m2
    }
}
// m1是静态函数，m2是实例函数
class A{
    public static void m1(){  System.out.println("A's m1"); }
    public void m2(){ System.out.println("A's m2"); }
}

class B extends A{
    public static void m1(){  System.out.println("B's m1"); }
    public void m2(){ System.out.println("B's m2"); }
}

class C extends B{
    public static void m1(){  System.out.println("C's m1"); }
```

```
    public void m2(){ System.out.println("C's m2"); }
}
```

- 在没有创建对象的情况下，静态方法无法访问非静态变量。
- 创建对象时new会自动调用构造函数，根据实参确定调用哪个构造函数。
- 当创建对象数组时，数组元素的缺省初值为null。

举个例子：

```
Circle[] circleArray = new Circle[10]; //这时没有构造Circle对象，只是构造数组
for(int i = 0; i < circleArray.length; i++) {
    circleArray[i] = new Circle( );    //这时才构造Circle对象，可使用有参构造函数
}
```

- 构造函数必须是**实例方法（无static）**，可为公有、保护、私有和包级权限。如果类未定义任何构造函数，编译器会自动提供一个不带参数的默认构造函数（缺省构造函数）。如果已自定义构造函数，则不会提供默认构造函数。
- 构造函数的递归嵌套举例：

```
class C {
    int x;
    String y;
    public C() {
        this(“1”);
        System.out.print(“one ”);
    }
    public C(String y) {
        this(1, “2”);
        System.out.print(“two ”);
    }
    public C(int x, String y) {
        this.x = x;
        this.y = y;
        System.out.print(“three ”);
    }
    public static void main(String[] args) {
        C c = new C();
        System.out.println(c.x + “ ” + c.y);
    }
}
// 程序的输出结果是 three two one 1 2    先调用C()，在C()中调用C(String y)，在C(String y)中调用C(int x, String y)，接着再依次返回
```

- 实例常量是没有用static修饰的final变量。
- 静态常量是用static修饰的final变量。Math类中的静态常量PI定义为：

```
public static final double PI = 3.14159265358979323846
```

- final修饰实例方法时，表示该方法不能被子类覆盖(Override)。非final实例方法可以被子类覆盖（见继承）。final修饰静态方法时，表示该方法不能被子类隐藏(Hiding)。非final静态方法可以被子类隐藏。

- 用final关键字修饰一个类表示：该类不可以被继承，为最终类，如Math类，此final类的成员方法没有机会被覆盖，默认都是final的。在设计类时候，如果这个类不需要有子类，类的实现细节不允许改变，并且确信这个类不会再被扩展，那么就设计为final类。
- 方法重载：同一个类中、或者父类子类中的多个方法具有相同的名字，但这些方法**具有不同的参数列表**(不含返回类型，即无法以返回类型作为方法重载的区分标准）。
- 方法覆盖和方法隐藏：发生在父类和子类之间，前提是继承。子类中定义的方法与父类中的方法**具有相同的方法名字、相同的参数列表、相同的返回类型**（也允许子类中方法的返回类型是父类中方法返回类型的子类）。方法覆盖：实例方法，方法隐藏：静态方法。
- 类的成员变量(实例变量和静态变量)的作用域是整个类，与声明的位置无关。

java的继承

- Java的继承都是公有继承，一个类只能有一个父类。
- 实例初始化模块只有在创建类的实例时才会调用。一个类可以有多个实例初始化块，对象被实例化时，模块按照在类中出现的顺序执行，构造函数最后运行。
  举个例子:

```
public class Book{
  private int id = 0;        //执行次序：1  定义并初始化类的实例变量等价于实例初始化块
  public Book(int id){       //执行次序：4
    this.id = id
  }
  {
     //实例初始化块            //执行次序：2
  }
  {
     //实例初始化块            //执行次序：3
  }
}
```

- 静态初始化模块是由static修饰的初始化模块{ }，只能访问类的静态成员，并且在JVM的Class Loader将类装入内存时调用。（**类的装入和类的实例化是两个不同步骤，首先是将类装入内存，然后再实例化类的对象**）。
- **初始化模块执行顺序**如下：
  - 父类的静态初始化模块。
  - 子类的静态初始化模块。
  - 父类的实例初始化模块。
  - 父类的构造函数。
  - 子类的实例初始化模块。
  - 子类的构造函数。
  - 举个例子:

```
public class InitDemo{
    InitDemo(){
        new M();
    }
    public static void main(String[] args){
```

```
        System.out.println("(1) ");  // 静态初始化块执行完毕，JVM就会调用main方法
        new InitDemo();
    }
    {
        System.out.println("(2) ");
    }
    static{
        System.out.println("(0) ");
    }
}
class N{
    N(){    System.out.println("(6) "); }
    {
        System.out.println("(5) ");
    }
    static {
        System.out.println("(3) ");
    }
}
class M extends N{
    M(){    System.out.println("(8) "); }
    {
        System.out.println("(7) ");
    }
    static {
        System.out.println("(4) ");
    }
}
```

- 利用super可以显式调用父类的构造函数。必须是子类构造函数的第1条且仅1条语句(先构造父类)。如果子类构造函数中没有显式地调用父类的构造函数，那么将自动调用父类不带参数的构造函数。
- 静态方法和静态初始化块里不能使用super和this。
- 如果父类没有无参构造函数，那么子类构造函数里若调用父类无参构造函数就会编译出错。
  举个例子：

```
class Base {
    public Base(String s) {
        System.out.print("B");
    }
}
public class Derived extends Base {
    public Derived (String s) {  // Base没有缺省构造函数
        System.out.print("D");
    }
    public static void main(String [] args) {
        new Derived ("C");
    }
}
```

- equals用于判断一个对象同另一个对象的所有成员内容是否相等。覆盖时应考虑：
  - 对基本类型数值成员。直接使用==判断即可。
  - 对引用类型变量成员。则需要对这些变量成员调用equals判断，不能用==（如**String**）。

    - 覆盖equals函数，首先用instanceof检查参数的类型是否和当前对象的类型一样。
- 要实现一个类的clone方法，首先这个类需要实现Cloneable接口，否则会抛出**CloneNotSupportedException**异常。
    - 还要公有覆盖clone方法，即Object类里clone方法是保护的，子类覆盖这个方法时应该提升为public。
    - 要克隆的对象可能包含基本类型数值成员或引用类型变量成员，对于基本类型数值成员使用=赋值即可，对于引用类型成员则需要进一步嵌套调用该成员的克隆方法进行赋值。
- 用子类对象作为实参传给方法中的父类型形参没有问题。

抽象类和接口

- Java可定义不含方法体的方法，其方法体由子类根据具体情况实现，这样的方法称为抽象方法(abstract method)，包含抽象方法的类必须是抽象类(abstract class)。只要类C有一个未实现的方法（自己定义的或继承的），就是抽象类。但是，一个不包含任何抽象方法的类，也可以定义成抽象类。注意，**抽象类不能被实例化。**
- 一个抽象类型引用变量可以指向具体子类的对象。
- 抽象类可以定义构造函数，并可以被子类调用（通过super）。 抽象类可以定义变量（实例或静态）、非抽象方法并被子类使用。抽象类的父类可以是具体类：自己引入了抽象方法。例如，具体类Object是所有类的祖先父类。
- 接口是公共静态常量和公共抽象实例方法的集合。
    - 接口不能定义构造函数。
    - 接口之间可以多继承，一个类可实现多个接口。
    - 和抽象类一样，不能实例化。
- 接口中的所有数据字段隐含为public static final。接口体中的所有方法隐含为public abstract。

java异常

- 在java中对于程序可能出现的必检异常，要么用**try...catch**语句捕获并处理它，要么使用**throws**语句抛出它，由上一级调用者来处理。
- 在java中异常分为**必检异常**和**非必检异常**二种类型，其中表达式**10/0**会抛出**非必检类型异常**，打开一个**不存在的文件**会抛出**必检类型异常**，通过**空引用调用实例方法**会抛出**非必检类型异常**，**数组越界访问**会抛出**非必检类型异常**，用 `throw` 语句抛出一个自定义的 `Exception` 子类异常：这取决于自定义异常是否是 `RuntimeException` 的子类。如果是，那么它是非必检异常（unchecked exception）。如果不是，并且是 `Exception` 的直接子类或 `Throwable` 的其他直接子类，那么它是必检异常（checked exception）。

java泛型

- 主要优点是在编译时而不是运行时检测出错误。
- **ArrayList**定义了一个带类型形参的泛型类，类型参数E是形参。**ArrayList**是一个参数化类型(**实例类型**)，其中**String**作为一个具体类型（实参）传递给形参**E**。 这里借用了术语"实例"，不是指对象，而是一个具体的类型。类型实参String传递给类型形参E是发生在编译时（不是运行时）。因此，对于下面的语句，编译器会用String代替E，对代码进行类型检查。

```
1 ArrayList<String> list = new ArrayList<>();//用实例类型ArrayList<String> 声明引用变量list
2 list.add("China");//编译器会根据类型实参String检查传入add方法的对象类型是否匹配，否则报错
```

- 泛型函数的一个例子：

```
public class Max {
    public static <E extends Comparable<E>> E findMax(E o1, E o2){
        return (o1.compareTo(o2) > 0)?o1:o2;
    }
}
// E extends Comparable<E>>指定类型E必须实现Comparable接口，而且接口比较对象类型必须是E
// 注意：在指定受限的类型参数时，不管是继承父类还是实现接口，都用extends
```

- java中类型T可以使用的地方：

```
对于泛型类class A<T> { … }，T在A类里可以用作不同的地方，在A类类体内，下面语句正确的有A B D G。
A.  T x;
B.  T m1() {return null;}
C.  static T y;
D.  void m2(T i) {}
E.  static T s1() {return null;}
F.  static void s2(T i) {}
G.  static <T1> void s3(T1 i, T1 j){}
// 就算A是泛型类，也无法用来声明静态数据，如果要用于静态方法，必须将类型参数<T>置于返回类型之前
```

- java泛型使用注意点：
  - 泛型的类型参数只能是类类型（包括自定义类），不能是基本类型。
  - 不能对泛型的具体实例类型使用instanceof操作，如 o instanceof ArrayList，否则编译时会出错。
  - 不能创建一个泛型的具体实例类型的数组，如 new ArrayList[10]，否则编译时会出错。
- 数组的协变性是指：如果类A是类B的父类，那么A[]就是B[]的父类。
  举个例子：

```
class Fruit{}
class Apple extends Fruit{}
class Jonathan extends Apple{} //一种苹果
class Orange extends Fruit{}

//由于数组的协变性，可以把Apple[]类型的引用赋值给Friut[]类型的引用
Fruit[] fruits = new Apple[10];
fruits[0] = new Apple();
fruits[1] = new Jonathan();

try{
    //下面语句fruits的声明类型是Fruit[]因此编译通过，但运行时将Fruit转型为Apple错误
    fruits[2] = new Fruit();//运行时抛出异常java.lang.ArrayStoreException，这是数组协变性导致的问题
}catch(Exception e){
    System.out.println(e);
}
  //数组是在运行时才去判断数组元素的类型约束；
```

```
18    //而泛型正好相反，在运行时，泛型的类型信息是会被擦除的，编译的时候去检查类型约束
```

- 通配泛型
  - 上界通配泛型

```
1  ArrayList<? extends Fruit> list = new  ArrayList<Apple>();
2  list.add(new Apple()); list.add(new Fruit()); //编译都报错
3  list.add(null); //可加入null
4  Fruit f = list.get(0);
5  // 带<? extends>类型通配符的泛型类不能往里存内容（不能set），只能读取（只能get）
6
7  //注意
8  ArrayList<? extends Fruit> list1 = new ArrayList<Fruit>(); //=号右边，如果是
   Fruit，可以不写，等价于new ArrayList<>()
9  ArrayList<? extends Fruit> list2 = new ArrayList<Apple>();  //=号右边，如果
   是Fruit的子类，则必须写
```

  - 下界通配泛型

```
1  //采用下界通配符 ？super T 的泛型类引用，可以指向所有以T及其T的父类型为类型参数的实
   例类型
2  ArrayList<? super Fruit> list = new ArrayList<Fruit>(); //这时new后边的
   Fruit可以省略
3  ArrayList<? super Fruit> list2 = new ArrayList<Object>(); //允许，Object
   是Fruit父类
4  ArrayList<? super Fruit> list3 = new ArrayList<Apple>(); //但是不能指向
   Fruit子类的容器
5
6  list.add(new Fruit());  //OK
7  list.add(new Apple());  //OK
8  list.add(new Jonathan());   //OK
9  list.add(new Orange()；  //OK
10 list.add(new Object()); //添加Fruit父类则编译器禁止，报错
11
12 // 从list里get数据只能被编译器解释成Object
13 Object o1 = list.get(0); //OK
14 Fruit o2 = list.get(0);  //报错，Object不能赋给Fruit，需要强制类型转换
```

  - 总的理解

```
1  // ArrayList<? extends Fruit> list可指向
   ArrayList<Fruit>|ArrayList<Apple>|ArrayList<Jonathan>|
   ArrayList<Orange>|…
2  // 一个ArrayList<Fruit>容器可以加入Fruit、Apple、Jonathan、Orange
3  // 编译器禁止ArrayList<? extends Fruit>类型的list添加元素
```

- 实现带泛型参数的类型工厂

```
1  public class ObjectFactory<T> {
2      private Class<T> type; // 定义私有数据成员，保存要创建的对象的类型信息
3      public ObjectFactory(Class<T> type) {
4          this.type = type;
5      }
6      public T create() {
```

```
        T o = null;
        try {
            o= type.newInstance(); //对象工厂的create方法负责产生一个T类型的对象，利用newInstance
        } catch (InstantiationException | IllegalAccessException e) {
            e.printStackTrace();
        }
        return o;
    }
}

public class Test {
    public static void main(String[] args) {
        //首先创建一个负责生产Car的对象工厂，传进去需要创建对象的类的Class信息
        ObjectFactory<Car> carFactory = new ObjectFactory<Car>(Car.class);
        Car o = carFactory.create();  //由对象工厂负责产生car对象
        System.out.println(carFactory.create().toString());
    }
} //以Car.class为参数去构造一个ObjectFactory<Car>类型的对象工厂，再调用对象工厂的create方法，一定会返回Car对象。

public class Car {
    private String s = null;
    public Car() {
        s = "Car";
    }
    public String toString() {
        return s;
    }
}
```

- 获取class对象

```
class Person{  }

class Employee extends Person{}

class Manager extends Employee{ }

try {
            Class clz = Class.forName("ch13.Manager"); //参数是类完全限定名字符串
            System.out.println(clz.getName()); //产生完全限定名ch13.Manager
            System.out.println(clz.getSimpleName()); //产生简单名Manager

            Class superClz = clz.getSuperclass(); //获得直接父类型信息
            System.out.println(superClz.getName()); //产生完全限定名ch13.Employee
            System.out.println(superClz.getSimpleName()); //产生简单名Employee
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        }

```

```
Class clz = Manager.class;  // Manager.class得到Manager的Class对象，赋给引用clz
System.out.println(clz.getName()); //产生完全限定名ch13.Manager
System.out.println(clz.getSimpleName()); //产生简单名Manager

Object o = new Manager();
Class clz = o.getClass();
System.out.println(clz.getName()); //产生完全限定名ch13.Manager
System.out.println(clz.getSimpleName()); //产生简单名Manager
```

java的某些类

- **Math是final类**：在**java.lang.Math**中，所有数学函数都是静态方法。
- **a + (int) (Math.random( )*b)**生成[a, b)之间的整数
- **(char)(ch1+(int)(Math.rabdom()*(ch2-ch1+1)))** 可以生成任意2个字符ch1和ch2（ch1<ch2）之间的随机字符。
- **String**类
  - **.length()**方法返回字符串长度。
  - **.charAt(index)**方法提取某个特定的字符，index取**[0, s.length() - 1]**
  - **String s3 = s1 + s2**等价于**String s3 = s1.concat(s2)**，该操作返回一个新的字符串。
  - **.substring(a, b)**返回**s[a, b - 1]**的新字串
  - **字符数组和字符串间的转换**

```
String s = “Java”;
char[ ] charArray = s.toCharArray( );
String.valueOf( new char[ ] {‘J’,‘a’,‘v’,‘a’} );
String.valueOf(2.34);  //2.34转“2.34”
String s2 = String.valueOf(true); //“true”
Integer intValue = Integer.valueOf(str);
int intValue = Integer.parseInt(intString);

//回文
public class CheckPalindrome {
    public static boolean isPalindrome(String s) {
    // The index of the first character in the string
    int low = 0;
    // The index of the last character in the string
    int high = s.length( ) - 1;
    while (low < high) {
        if (s.charAt(low) != s.charAt(high)) return false; // Not a palindrome
        low++;
        high--;
    }
    return true; // The string is a palindrome
    }
}
```

- **StringBuffer**类

```
StringBuffer sb = new StringBuffer();// 可以以字符串构造
```

```
sb.append("Hello, ");
sb.append("World!");
System.out.println(sb); // 输出: Hello, World!

sb.insert(0, "Hi ");
System.out.println(sb); // 输出: Hi Hello, World!

sb.delete(0, 3);
System.out.println(sb); // 输出: Hello, World!

sb.replace(0, 5, "Hi");
System.out.println(sb); // 输出: Hi World!

String str = sb.toString();
System.out.println(str); // 输出: Hi World!
// 还有.reverse()方法
```

java多线程总结

- 有两种方法可以实现同一个或多个线程的运行：(1) 定义Thread类的子类并覆盖run方法；(2) 实现接口Runnable的run方法。
- 线程同步
  - 假设一个类有多个用synchronized修饰的同步实例方法, 如果多个线程访问这个类的同一个对象，当一个线程获得了该对象锁进入到其中一个同步方法时，这把锁会锁住这个对象所有的同步实例方法.
  - 假设一个类有多个用**synchronized**修饰的**同步实例方法，如果多个线程访问这个类的**不同对象，**那么**不同对象的synchronized锁不一样，每个对象的锁只能对访问该对象的线程同步**.
  - 如果采用Lock锁进行同步，一旦Lock锁被一个线程获得，那么被这把锁控制的所有临界区都被上锁，这时所有其他访问这些临界区的线程都被阻塞。
  - 如果一个类采用Lock锁对临界区上锁，**而且这个**Lock锁也是该类的实例成员（见ResourceWithLock的里的lock对象定义），那么这个类的二个实例的Lock锁就是不同的锁**，下面的动画演示了这种场景：对象o1的Lock锁和对象o2的Lock锁是不同的锁对象。

```
 class Resource{ //共享资源类，这个类的实例被多个线程访问
    private int value = 0;    //多个线程会同时对这个数据成员读写
    public int getValue(){return  value;}
    public synchronized void inc(int amount)  {
        System.out.print("\nThread " + Thread.currentThread().getId() + " 进入inc: ");
        int newValue = value + amount;
        try{ Thread.sleep(5); }  catch(InterruptedException e){ }
        value = newValue;
        System.out.print("-->Thread " + Thread.currentThread().getId() + " 离开inc.");
    }
    public synchronized void dec(int amount)  {
        System.out.print("\nThread " + Thread.currentThread().getId() + " 进入dec: ");
        int newValue = value - amount;
        try{ Thread.sleep(2); } catch(InterruptedException e){  }
        value = newValue;
```

```
        System.out.print("-->Thread " + Thread.currentThread().getId() + " 离开dec.");
    }
}

class IncTask implements Runnable{
    private Resource r = null; //要访问的对象
    private int amount = 0; //每次增加量
    private int loops = 0;  //循环次数
    public IncTask(Resource r,int amount,int loops){
        this.r = r; this.amount = amount; this.loops = loops;
    }
    public void run() {
        for(int i = 0; i < loops; i++) { r.inc(amount); }
    }
}
class DecTask implements Runnable{
    private Resource r = null; //要访问的对象
    private int amount = 0; //每次减少量
    private int loops = 0;  //循环次数
    public DecTask(Resource r, int amount, int loops){
        this.r = r;this.amount = amount;this.loops = loops;
    }
    public void run() {
        for(int i = 0; i < loops; i++) {  r.dec(amount); }
    }
}
public class T1 {
    public static void main(String[] args){
        int incAmount = 10;
        int decAmount = 5;
        int loops = 100;

        Resource r = new Resource();
        Runnable incTask = new IncTask( r, incAmount, loops );
        Runnable decTask = new DecTask( r, decAmount, loops );

        ExecutorService es = Executors.newCachedThreadPool();
        es.execute(incTask); es.execute(decTask);
        es.shutdown();
        while(!es.isTerminated()){  }

        int correctValue = (incAmount - decAmount)  * loops;
        System.out.println("\nThe value: " +  r.getValue() + ", correct value: " + correctValue);
    }

}
```