方法重载与静态成员

引例

子类往父类会自动转，父类往子类需要进行强制类型转换

```java
Employee e = new Manager();
e.getDepartment();
Manager m = (Manager)e;
m.getDepartment();
```

在本例中，由于`Manager`继承了`Employee`，且只有`Manager`类定义了`getDepartment()`方法，那么`e.getDepartment()`将无法执行。

在Java中，一个对象可以赋值给一个其父类或子类的引用变量。在这个例子中，`Manager`对象被赋值给了`Employee`类型的引用变量`e`。由于`Employee`类本身没有定义`getDepartment()`方法，而`e`被声明为`Employee`类型，所以编译器只允许调用`Employee`类中定义的方法。因此，直接通过`e.getDepartment()`将无法编译通过。

然而，如果将`e`强制转换为`Manager`类型，如`Manager m = (Manager)e;`，那么就可以调用`getDepartment()`方法，因为此时`m`是一个`Manager`类型的引用变量，可以调用`Manager`类中定义的方法。

综上所述，`e.getDepartment()`将无法执行，而`m.getDepartment()`可以执行并返回调用`Manager`类中定义的`getDepartment()`方法的结果。

多态的问题： 如何判断一个变量所实际引用的对象类型？

对象类型识别

为了保证引用变量的类型转换不会在运行时引发异常，影响程序的运行，通常先要判断对象的类型。

`instanceof`**操作符**：判断对象类型，返回`true`或`false`

- `Object instanceof Class`
- `Sub_instance instanceof Sup_class: true`
  子类继承父类，故判断子类对象是否属于父类也会返回`true`

```
Manager m = new Director();
m instanceof Director; // ture
m instanceof Manager; // true
```

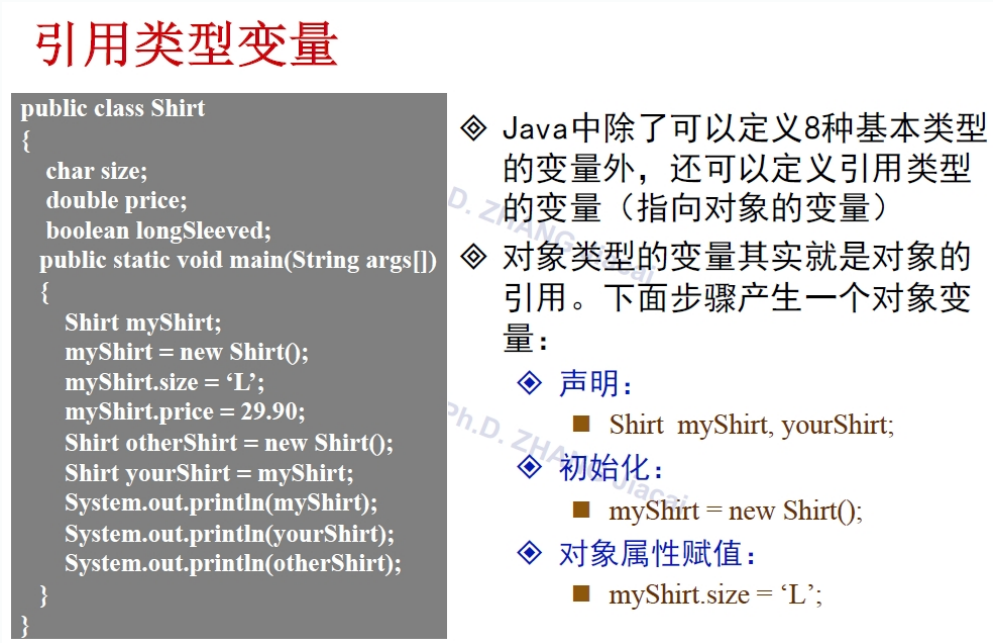
JAVA

```
public class InstanceofTest{
        public stati void main (String[] args){
                Employee e1 = new Manager();
                Employee e2 = new Employee("gzhu");
                System.out.println("e1 is instanceof manager:" + (e1 instanceof
Manager)); // ture, `e1`是`Manager`类的实例
                System.out.println("e1 is instanceof employee:" + (e1 instanceof
Employee)); // true, 尽管`e1`是`Manager`的实例，但由于`Manager`继承自`Employee`，所以
`e1`也是`Employee`的一个实例。
                System.out.println("e2 is instanceof manager:" + (e2 instanceof
Manager)); // false, `e2`只是`Employee`类的实例，并不是`Manager`类的实例
                System.out.println("e2 is instanceof employee:" + (e2 instanceof
Employee)); // true
        }
}
```

判断相等(对象内容比较)

在Java中，用类作为抽象数据类型声明的变量是引用变量，该变量指向一个对象在内存中的位置

引用类型变量



引用类型变量（续）

```
public class Shirt
{
   char size;
   double price;
   boolean longSleeved;
  public static void main(String args[])
  {
     Shirt myShirt;
     myShirt = new Shirt();
     myShirt.size = 'L';
     myShirt.price = 29.90;
     Shirt otherShirt = new Shirt();
     Shirt yourShirt = myShirt;
     System.out.println(myShirt);
     System.out.println(yourShirt);
     System.out.println(otherShirt);
  }
}
```

◈ 对象类型的变量代表一个对象，本身的存储区域没有存放对象。对象存储在其它位置，变量代表的是对象的地址（就象C++中的指针）

原始变量	内存中的值
a	2
myShirt	0x34a012
yourShirt	0x34a012
otherShirt	0x99f311

此处必须要注意：对象类型的变量里面存储的只是对象的地址!!

方法重载与静态成员 > 判断相等

成员变量作为引用类型会自动初始化为`null`，但局部变量需要自行初始化（**数组除外**）

引用型变量只支持**有限的逻辑判断**：

- 相等判断（是否*同一个对象*的引用）：== or !=
  - 基本类型变量(int、char等8种)，`\==`判断变量值是否相等
  - theObject == null
  - otherObject != theObject
  - *对于引用类型，不比较对象的内容，只判断地址是否相同*

例子(内置字符串)：

```java
String s1 = "Welcome to Java";
String s2 = new String("Welcome to Java");
String s3 = "Welcome to Java";

Systeem.out.println("s1 == s2 is" + (s1 == s2)); // false
Systeem.out.println("s1 == s3 is" + (s1 == s3)); // true
```

内置(interned)字符串

◈ 由于字符串对象是不变的，可以提高程序频繁使用字符串的效率

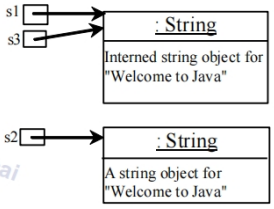
◈ JVM对于同样字符序列保存为唯一的样本实例，这个样本实例称为内置的字符串

```
String s1 = "Welcome to Java";

String s2 = new String("Welcome to Java");

String s3 = "Welcome to Java";

System.out.println("s1 == s2 is " + (s1 == s2));
System.out.println("s1 == s3 is " + (s1 == s3));
```



s1 == s is false
s1 == s3 is true

如果不用new来创建，而是用初始值，那就用内置的字符串对象，没有新的字符串创建

在 Java 中，创建字符串有两种基本方式，它们在内存使用和对象的处理上有所不同。理解这一点需要先了解 Java 字符串池（String Pool）的概念。

1. 字符串字面量（例如 `String s1 = "abc";`）：
   - 当你使用**字符串字面量**创建一个字符串时，JVM 首先检查字符串池中是否已经**存在相同内容**的字符串。
   - 如果存在，它不会创建新的对象，而是**直接返回对池中现有对象的引用**。
   - 如果不存在，它在池中创建一个新字符串，然后返回这个新对象的引用。
   - 这种方式是为了提高效率和减少内存开销。
2. 使用 `new` 关键字（例如 `String s2 = new String("abc");`）：
   - 使用 `new` 关键字会**强制在堆内存（Heap）上创建一个新的字符串**对象，即使字符串池中已经存在相同内容的字符串。
   - 这就意味着，即使字符串内容相同，使用 `new` 创建的字符串也会有不同的内存地址，即它**是一个完全独立的对象**。

因此，对于 `String s1 = "abc";` 和 `String s2 = new String("abc");`：

- `s1` 可能引用字符串池中的一个字符串对象（如果该内容之前已经被创建过）。
- `s2` 总是创建一个新的字符串对象在堆上，不管字符串池中是否已经存在内容相同的字符串。

这种差异在比较字符串时尤为重要。使用
`\==` 比较字符串时，它比较的是对象的引用（内存地址），而不是内容。因此即使两个字符串内容相同，如果一个来自字符串池，另一个通过 `new` 创建，使用 `\==` 比较时会返回 `false`。相反，应该使用 `equals` 方法来比较字符串的内容。例如：

```java
String s1 = "abc";
String s2 = new String("abc");

System.out.println(s1 == s2); // false, 因为引用不同
System.out.println(s1.equals(s2)); // true, 因为内容相同
```

总结，`String s1 = "abc";` 和 `String s1 = new String("abc");` 在内存分配和对象引用上存在差异，这在处理字符串对象时是非常重要的考虑因素。

equals方法

如何像基本类型变量那样判断引用类型变量内容是否相同？
`equals`方法：判断对象内容是否相等

`Object`类提供方法`equals(Object o)`判断对象内容是否相等：

- `boolean equals(Object obj)`，判断对象内容是否相同
- `equals`方法在`Object`类中的实现等同于==，因此子类中如果直接使用父类的`euqals()`方法等同于==，父类中不知道子类的成员结构，无法定义`equals()`的细节
- 子类常常要根据自身的成员结构，*重写`equals(Object obj)`* 方法以判断对象的内容是否相同

equals方法重写

- `Object`类是所有类的根，类声明中缺省继承的都是`Object`类
  - `public class Employee{}`与`public class Employee extends Object{}`等同

- Java中所有类都继承了Object类的方法，这些方法可以在子类中进行重写。Object类提供了很多方法可以重写

JAVA

```java
public class MyDate{
        private int day, month, year;
        public MyDate(int d, int m, int y){
                day = d; month = m; year = y;
        }
        public boolean equals(Object o){
                boolean result = false;
                if((o != null) && (o instanceof MyDate)){
                        MyDate d = (MyDate)o;
                        if((day == d.day) && (month == d.month) && (year == d.year)){
                                result = true;
                        }
                }
                return result;
        }
        public static void main(String[] args){
                MyDate one = new MyDate(1,10,2006);
                MyDate another = new MyDate(1,10,2006);
                System.out.println(one == another);  // false
                System.out.println(one.equals(another)); // true
        }
}
```

Q1：为什么要将o强制转换为MyDate类型?

> 在Java中，`equals`方法接收一个`Object`类型的参数。因此，当我们在这个方法中接收到一个对象时，我们**并不直接知道它是否是`MyDate`类的实例**。为了能够对它进行像`MyDate`对象那样的处理（例如，访问它的`day`，`month`，和`year`字段），我们需要首先确认它确实是一个`MyDate`对象。
> 强制类型转换`(MyDate)`的作用是将对象从它的原始类型（在这里是`Object`）转换为`MyDate`类型。这样，我们就可以访问该对象的`MyDate`特定属性和方法。
> 但是，在进行强制类型转换之前，我们需要确保该对象确实是一个`MyDate`实例。这就是`instanceof`操作符的用途。如果对象不是`MyDate`的实例，那么尝试进行强制类型转换将会抛出一个`ClassCastException`。
> 所以，将`o`强制转换为`MyDate`类型是为了能够像处理`MyDate`对象那样处理它，但前提是我们已经确认了它确实是一个`MyDate`对象。

Q2：我传入的another是MyDate类型，为什么我不能直接`equals(MyDate o)`?

> 虽然`another`确实是`MyDate`类型，**但`equals`方法的定义要求它接收一个`Object`类型的参数**。这是因为在Java中，所有的类都直接或间接地继承自`Object`类，所以`equals`方法定义在`Object`类中，其参数也是`Object`类型。
> 当你重写或实现一个方法时（比如`equals`），你**必须使用与父类方法相同的参数签名**。这就是为什么`equals`方法必须接收一个`Object`类型的参数，而不是`MyDate`类型。
> 这样的设计允许`equals`方法能够处理任何类型的对象，不仅仅是`MyDate`类型。在`equals`方法内部，你可以检查传入的对象是否是`MyDate`类型，并进行相应的处理.
> 所以，即使你知道传入的`another`是`MyDate`类型，你仍然需要将参数类型定义为`Object`，并在方法内部进行类型转换和检查。

Q3: `if((day == d.day) && (month == d.month) && (year == d.year))`Java是怎么知道day是当前对象（例如one）的day字段的?

> 在Java中，当你在一个类的方法内部访问一个变量，Java会首先**查找该方法内部是否有局部变量或参数与该变量名相同**。如果在方法内部没有找到，Java会**继续查找类的实例变量是否有与该变量名相同的变量**。
> 在`equals`方法中，我们并没有定义名为`day`的局部变量或参数，所以当我们在该方法内部简单地使用`day`这个变量时，Java会**将其解析为当前对象的实例变量**，即`one`的`day`字段。
> 同样地，当我们使用`d.day`时，`d`是另一个`MyDate`对象（例如`another`），所以`d.day`表示的是`another`的`day`字段。
> 总之，Java通过解析变量名和查找作用域来确定你引用的是哪个变量或字段。

异构集合

- 同类的对象的集合称为同构集合

```java
MaDate[] date = new MyDate[2];
date[0] = new MyDate(26,7,2001);
date[1] = new MyDate(15,10,2023);
```

- 不同类的对象的集合称为异构集合

```java
Employee[] staff = new Employee[3];
staff[0] = new Manager();
staff[1] = new Employee();
staff[2] = new Engineer();
```

- 对于异构集合中的对象，可以看作`Employee`集合，比如按`salary`排序。
- Java中`Object`类是所有类的父类，`Object`数组可放所有对象
- 原始类型的数据经包装为类后，也可以看作`Object`类的子类

异构集合的排序问题

要求编写一个对数组中元素进行排序和方法，该方法能够对整型数组、浮点数数组和字符串数组排序

- 方法1：**方法重载**
  - 对每种数据类型，各写一个`sortArray()`函数
    - `sortArray(int[] a); sortArray(double[] a); sortArray(String[] a)`
  - 缺点是代码重用性太差

JAVA

```java
import java.util.Arrays;

public class Main {
        // 对整型数组进行排序
        public static void sortArray(int[] a) {
                Arrays.sort(a);
        }
        // 对浮点数数组进行排序
        public static void sortArray(double[] a) {
                Arrays.sort(a);
        }
        // 对字符串数组进行排序
        public static void sortArray(String[] a) {
                Arrays.sort(a);
        }

        public static void main(String[] args) {
                int[] intArray = {3, 1, 4, 1, 5, 9};
                sortArray(intArray);
                System.out.println("Sorted int array: " + Arrays.toString(intArray));

                double[] doubleArray = {3.2, 1.1, 4.5, 1.3, 5.6, 9.0};
                sortArray(doubleArray);
                System.out.println("Sorted double array: " +
Arrays.toString(doubleArray));

                String[] strArray = {"Java", "Python", "C++", "JavaScript", "Go"};
                sortArray(strArray);
                System.out.println("Sorted string array: " +
Arrays.toString(strArray));
        }
}
```

- 方法二：**方法参数利用多态**
  - 方法参数类型为`Object`类，`sortArray(Object[] a)`
  - 但要先将Object对象*强制类型转换*为各种数据类型后才能排序
    - Object和各种数据类型间强制转换
    - 代码中包含类型判断和各种类型数据的排序实现
  - 缺点：必须**保证数组中元素类型**，否则类型转换有风险

JAVA

```java
import java.util.Arrays;

public class Main {
        public static void sortArray(Object[] a) {
                if (a instanceof Integer[]) { // 注意这里是Integer数组哎
                Arrays.sort((Integer[]) a);
                } else if (a instanceof Double[]) {
                Arrays.sort((Double[]) a);
                }
                else if (a instanceof String[]) {
                Arrays.sort((String[]) a);
                } else {
                throw new IllegalArgumentException("Unsupported array type");
                }
        }

        public static void main(String[] args) {
                Integer[] intArray = {3, 1, 4, 1, 5, 9};
                sortArray(intArray);
                System.out.println("Sorted int array: " + Arrays.toString(intArray));
                Double[] doubleArray = {3.2, 1.1, 4.5, 1.3, 5.6, 9.0};
                sortArray(doubleArray);
                System.out.println("Sorted double array: " +
Arrays.toString(doubleArray));
                String[] strArray = {"Java", "Python", "C++", "JavaScript", "Go"};
                sortArray(strArray);
                System.out.println("Sorted string array: " +
Arrays.toString(strArray));
        }
}
```

- 方法三：**泛型技术**
  - *类型参数化*（类型形参与类型实参）
    - **变量参数化**就是*定义的时候不指定变量值，只放一个变量占位符*，代表这个地方有一个变量，使用时再指定具体值
    - **类型参数化**就是定义的时候不指定类型，只放一个类型占位符，代表这个地方有一个数据类型，使用时再指定具体类型的对象
- 泛型技术与方法参数类比
  - 方法中看到形式参数`method(int x)`，`x`是形参，`x` 只是占位符，表示实际应用中此处传入整型值，`method(3)`，这个3就是实参
  - 泛型提出**类型参数**的概念，声明中用类型形参，实际应用中传入类型实参

JAVA

```java
import java.util.ArrayList;
import java.util.Collections;
public class Main {
        public static <T extends Comparable<T>>
        void sortArrayList(ArrayList<T> list) {
                Collections.sort(list);
        }
        public static void main(String[] args) {
                ArrayList<Integer> intList = new ArrayList<>();
                intList.add(3);
                intList.add(1);
                intList.add(4);
                intList.add(1);
                intList.add(5);
                intList.add(9);
                sortArrayList(intList);
                System.out.println("Sorted integer ArrayList: " + intList);

                ArrayList<Double> doubleList = new ArrayList<>();
                doubleList.add(3.2);
                doubleList.add(1.1);
                doubleList.add(4.5);
                doubleList.add(1.3);
                doubleList.add(5.6);
                doubleList.add(9.0);
                sortArrayList(doubleList);
                System.out.println("Sorted double ArrayList: " + doubleList);

                ArrayList<String> strList = new ArrayList<>(); strList.add("Java");
                strList.add("Python");
                strList.add("C++");
                strList.add("JavaScript");
                strList.add("Go");
                sortArrayList(strList);
                System.out.println("Sorted string ArrayList: " + strList);
        }
}
```

ArrayList类

- 数组可以存储任何类型的对象，但数组一旦创建，大小固定
- 数组列表则可以动态调整存储的对象的数量

java.util.ArrayList<E>	
+ArrayList()	Creates an empty list.
+add(o: E) : void	Appends a new element o at the end of this list.
+add(index: int, o: E) : void	Adds a new element o at the specified index in this list.
+clear(): void	Removes all the elements from this list.
+contains(o: Object): boolean	Returns true if this list contains the element o.
+get(index: int) : E	Returns the element from this list at the specified index.
+indexOf(o: Object) : int	Returns the index of the first matching element in this list.
+isEmpty(): boolean	Returns true if this list contains no elements.
+lastIndexOf(o: Object) : int	Returns the index of the last matching element in this list.
+remove(o: Object): boolean	Removes the element o from this list.
+size(): int	Returns the number of elements in this list.
+remove(index: int) : boolean	Removes the element at the specified index.
+set(index: int, o: E) : E	Sets the element at the specified index.

泛型

基本格式:

```java
ArrayList<String>cities = new ArrayList<String>();
ArrayList<String>cities = new ArrayList<>(); // 二者等价
```

- JDK1.5之后支持了泛型技术，数据类型是形式类型，实际使用中用具体的数据类型来取代，本质是数据类型参数化
  - 泛型中用<>，形参中用()
  - ArrayList<E> 是泛型，E是泛型（形式类型）
  - 此处E指的是元素类型，不是值
- 泛型技术的特点
  - 使用中参数E可以是任意类型，使用灵活，不受类型限制
  - 类型安全，使用泛型后在编译器就可以检查类型
  - 消除类型强制转换，代码可读性强且降低运行出错概率

JAVA

```java
import java.util.ArrayList;
public class TestArrayList{
        public static void main (String[] args){
                ArrayList<String>cityList = new ArrayList<>();
                cityList.add("London");
                cityList.add("Denver");
                cityList.add("Paris");
                cityList.add("Miami");
                cityList.add("Seoul");
                cityList.add("Tokyo");
                System.out.println("List size? " + cityList.size());
                System.out.println("Is Miami in the list? " +
cityList.contains("Miami"));
                System.out.println("The list index of Denver? " +
cityList.indexOf("Denver"));
                System.out.println("Is the list empty? " + cityList.isEmpty()); //
Print false
                cityList.add(2, "Xian");
                cityList.remove("Miami");
                cityList.remove(1);
            System.out.println(cityList.toString());
                for (int i = cityList.size() - 1; i >= 0; i--)
                System.out.println(cityList.get(i) + " ");
                ArrayList<Circle> list = new ArrayList<>();
                list.add(new Circle (2));
                list.add(new Circle (3));
                System.out.println("The area of the circle? " +
list.get(0).getArea());
        }
}
```

数组与ArrayList异同，互相转化

Operation	*Array*	*ArrayList*
Creating an array/ArrayList	`String[] a = new String[10]`	`ArrayList<String> list = new ArrayList<>();`
Accessing an element	`a[index]`	`list.get(index);`
Updating an element	`a[index] = "London";`	`list.set(index, "London");`
Returning size	`a.length`	`list.size();`
Adding a new element		`list.add("London");`
Inserting a new element		`list.add(index, "London");`
Removing an element		`list.remove(index);`
Removing an element		`list.remove(Object);`
Removing all elements		`list.clear();`

```
String[] array = {"red", "green", "blue"};
ArrayList<String> list = new ArrayList<>(Arrays.asList(array));
```

```
String[] array1 = new String[list.size()];
list.toArray(array1);
```

Arrays是java.util包下的一个操作数组的类

方法重写与方法重载

方法重写(覆盖，overriding)

方法重写

- *新类继承父类*时（且只存在父子类之间），除了能添加成员外，还能改写父类中继承的行为，这就是方法重写或方法覆盖
- 如果子类中的方法名字、返回类型、参数与父类中方法都完全相同，那么新的方法改写（Overriding）了旧的方法。
- `super`用来代表父类，可以指定父类中的属性和方法
- 父类的**静态方法**只能被子类的静态方法重写，不能用子类的非静态方法覆盖
  - 子类隐藏父类的静态方法；与所属的类别绑定
  - 子类覆盖父类的实例方法；与所属的实例绑定
- 父类的**非静态方法**不能被子类重写为静态方法
- 父类的**私有方法**无法被重写
- *变量是静态绑定，由变量所声明的类型绑定*
- *实例方法是动态绑定*

- 子类继承父类时，除了能添加成员外，还能改写父类中继承来的行为，新的方法改写（Overriding）了旧的方法
  - 名称、参数、返回类型完全一样，访问权限不能缩小
  - 只存在于父子类之间，父类的静太方法只能被子类的静态方法重写，不能用子类的非静态方法来覆盖
  - 父类的非静态方法不有被子类重写为静态方法
  - 父类的私有方法无法被重写
- 调用时
  - 实例方法与引用变量所实际指向的对象方法绑定：动态绑定，运行时由Java虚拟机动态决定
  - 静态方法与引用变量所声明的类型的方法绑定：静态绑定，编译阶段就完成绑定
- 变量是静态绑定，由变量所声明的类型绑定
- 实例方法是动态绑定

例子：

```
public class Employee{
    ……
  public String getDetail(){
    return "Name:"+name+"\nSalary:"+salary;
  }
}
```

```
public class Manager extends Employee{
    ……
}
```

```
public class Director extends Manager{
    ……
  public String getDetail(){
    return super.getDetail()
          +"carAllowance" + carAllowance;
  }
}
```

super关键字

- 在子类中用`super`访问父类的成员

```java
class Base{
        String var = "Base's variable";
        void method(){
                System.out.println("call Base's method");
        }
}
public class Sub extends Base{
        String var = "Sub's variable";
        // 方法重写
        void method(){
                System.out.println("call Sub's method");
        }
        void test(){
                String var = "Local variable"; // 局部变量
                System.out.println("var is " + var);
                // var is Local variable
                System.out.println("this var is " + this.var);
                // 打印成员变量
                // this var is Sub's variable
                System.out.println("super var is " + super.var);
                // super var is Base's variable

                method(); // 调用Sub实例的method()方法
                this.method(); // 调用Sub实例的method()方法
                super.method();  // 调用在Base类中method()方法
        }
        public static void main(String args[]){
                new Sub.test(); // 这里可以直接new一个对象不用命名哎
        }
}
```

条件：

1. 子类继承父类方法
2. 子类和父类都定义了该方法
3. 方法*名字*、*返回类型*、*参数(类型、个数、顺序)* 和父类中方法都完全相同
4. 访问权限可以不同，*子类访问权限不能小于父类方法的访问权限*，不然会导致编译正常但是运行错误

例1

```java
public calss Parent{
        public void doSomething(){} // 父类方法为public
}
public calss Child extends Parent{
        private void doSomething(){} // 子类方法为private
}
public class doBoth{
        public void doOther(){
                Parent p2 = new Child();
                p2.doSomethind(); // 编译出错
        }
}
```

方法重载(overloading)

用于**同一个类**中几个不同方法完成同一任务，这方法仅仅因为参数不同，比如`print()`可以处理多种参数

- 对于不同类型的参数，有不同的处理
- Java语言允许**同一个方法名可以用于多个方法**，只要能够确定到底调用的是哪一个函数，比如根据参数的类型和参数的个数来确定

println(10L)、println("10")、println(10)

```
public void println(int i)
public void println(long l)
public void println(String s)
public void println()
```

◈ 同一个类的一个方法有多种实现：根据不同类型的参数，提供不同的实现方式

```
public class  ElectricalRepairShop {
    void repair(Computer  c){修理电脑}
    void repair(TV tv){修理电视机}
    void repair(AirConditioner ac){修理空调}
}
```

```
public static int max(int a,int b)
public static long max(long a,long b)
public static float max(float a,float b)
public static double max(double a,double b)
```

◈ 程序调用Math类的max()方法，Java虚拟机先判断给定参数的类型，然后决定到底执行哪个max()方法

```
//参数均为int类型，因此执行max(int a,int b)方法
Math.max(1,2);
//参数均为float类型，因此执行max(float a,float b)方法
Math.max(1.0F, 2.0F);
```

规则

- 调用语句的**参数列表必须足以区分适当的函数**
- 注意**有的类型可以自动转化**，float增容为double类型，父类兼容子类等
  - `public void testMethod(int x){}`
  - `public void testMethod(short y){}`
- 重载的方法间**可以有不同的返回类型**，但仅仅返回类型不同不足以区分方法，必须参数列表要有区别

总结：

- *方法名相同*
- 参数*类型，个数，顺序*至少一项不同
- *返回类型*可以不同
- *修饰符*可以不同
  注： JVM自动匹配能兼容参数的最小类型的参数

```java
public class TestClass{
        public void test1(int x){
                System.out.println("int");
        }
        public void test1(short x){
                System.out.println("short");
        }
        public static void main(String[] args){
                TestClass tc = new TestClass();
                tc.test1(12);  // 打印int
                tc.test1((byte)12);  // 打印short
        }
}
```

重写与重载的区别

- 重写的方法只能来自于父类，参数相同，返回类型相同
- 重载的方法可能继承自父类，也可能在同一个类中定义，参数不同

重写(*传入参数相同*)：

JAVA

```java
public class Test{
        public static void main(String[] args){
                A a = new A();
                a.p(10);  // 10.0
                a.p(10.0); // 20.0
        }
}
class B{
        public void p(double i){
                System.out.println(i * 2);
        }
}
// 方法重写
class A extends B{
        public void p(double i){ // 传入的参数和顺序均相同
                System.out.println(i);
        }
}
```

重载(*传入参数不同*):

JAVA

```java
public class Test{
        public static void main(String[] args){
                A a = new A();
                a.p(10);  // 10
                a.p(10.0); // 20.0
        }
}
class B{
        public void p(double i){
                System.out.println(i * 2);
        }
}
// 方法重载，传入参数类型变为int，但原有父类的方法仍然保留
class A extends B{
        public void p(int i){
                System.out.println(i);
        }
}
```

构造器重载

- 当对象创建并初始化时，可根据不同情况来实例化对象
  - 比如创建`Employee`对象，可能`name`，`salary`和`birthay`*三者都已知*，也可能仅*已知其中一二*
  - 注意构造器中的`this()`和`super()`，必须出现在**第一句**
- 构造器可能会**调用重载**的构造器

- 指在同一个类中存在若干个具有不同参数列表的构造器
- 创建该类对象的语句会自动根据给出的*实际参数的数目、类型和排列顺序***调用相应的构造函数**来完成对新对象的初始化操作
- 使用`this()`间接调用父类构造器

- 构造器中可能**显式调用**父类构造器
  - 不能用类名调用，只能用`super()`

```JAVA
public class Employee{
        private String name = "";
        private double salary;
        public Date birthDay;

        public Employee(String name, double salary, Date.birthday){
        this.name = name;
        this.salary = salary;
        this.birthDate = birthDate;
        }
        // 重载构造器
        public Employee(String name, double salary){this(name, salary, null);} // 调用一开始的构造器，在birthday 传入Null
        public Employee(String name, Date birthday){this(name, 10000, birthday);}
        public Employee(String name){this(name, 10000);} // 调用第二个构造器
}
```

更加隐式的构造器重载

例1：

```JAVA
class Base extends Object{}
class Sub1 extends Base {}

public class Sub2 extends Base {
        public void print(Base b){System.out.println("Base");}
        public void print(Sub1 s){System.out.println("Sub1");}
        public void print(Sub2 s){System.out.println("Sub2");}

        public static void main(String args[]){
                Sub2 sub = new Sub2();
                Base o = new Sub1();
                sub.print(sub); // Sub2
                sub.print(o);  // Base
        }
}
```

在 `main` 方法中：

- 创建了一个 `Sub2` 类的对象 `sub`。

- 创建了一个 `Sub1` 类的对象 `o`，并将其引用赋给了一个 `Base` 类型的变量。
- 调用了 `sub.print(sub)`，这会调用 `print(Sub2 s)` 方法，并输出 "Sub2"。
- 调用了 `sub.print(o)`，虽然 `o` 的实际类型是 `Sub1`，但因为它的引用类型是 `Base`，所以会调用 `print(Base b)` 方法，并输出 "Base"。

> **这种方法是基于静态绑定的**。静态绑定是指在编译时就已经确定了方法调用，与实际运行时对象的类型无关。在这个例子中，调用哪个 `print` 方法是在编译时根据引用类型确定的，而不是在运行时根据实际对象的类型确定的。因此，即使 `o` 的实际类型是 `Sub1`，但因为它的引用类型是 `Base`，所以会静态地绑定到 `print(Base b)` 方法。

特殊例子：

JAVA

```java
// example 1
class Base {}
public class Sub extends Base{
        public void print(Base b){ System.out.println("Base"); }
        public void print(Sub s){ System.out.println("Sub"); }
        public static void main(String args[]){
                Sub sub = new Sub();
                sub.print(null);
        }
}
// 打印 Sub
////////////////////////////////////////////////
// example 2
class Base {}
public class Sub {
        public void print(Base b){ System.out.println("Base"); }
        public void print(Sub s){ System.out.println("Sub"); }
        public static void main(String args[]){
                Sub sub = new Sub();
                sub.print(null);
        }
}
// 编译错误
// 可以修改为 Base b = null; sub.print(b);
```

对`example 1`的解释：

在Java中，方法重载（Overloading）是根据传入参数的数量、类型和顺序来决定调用哪个方法的。当你传递一个`null`值给`print`方法时，Java无法准确判断应该调用哪个版本的`print`方法，因为`null`**可以匹配任何引用类型**。

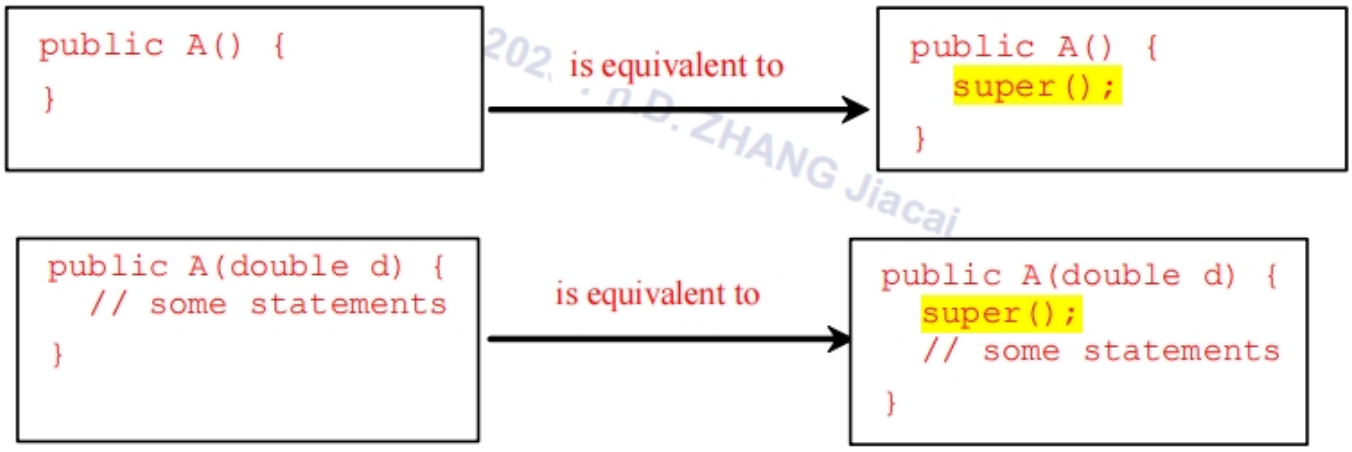
在这种情况下，Java遵循一个规则，即**“最特定（most specific）”**规则。如果多个方法都可以接受null作为参数，那么会**调用参数类型最特定的那个方法**。如果两个方法的参数类型一样特定，那么就不允许这样的重载，编译器会报错。

在`example 1`中，`print(Base b)`和`print(Sub s)`两个方法都可以接受null作为参数，但`Sub`是`Base`的子类，所以`Sub`**更特定**。因此，当你传递一个`null`给`print`方法时，会调用`print(Sub s)`这个版本的方法。所以程序的输出应该是“Sub”。
而在`example 2`中，当调用 `sub.print(null);` 时，Java编译器无法确定应该调用哪个 `print` 方法，因为 `null` 可以匹配任何引用类型，这里既可以是 `Base` 类型也可以是 `Sub` 类型。因此，编译器会报告这个方法调用为不明确（ambiguous），编译报错

注意：这个规则只适用于引用类型（类、接口和数组）。对于基本类型（如int、float、boolean等），null是无法接受的。

构造器的自动调用

- 如果构造器中既没有调用**重载的构造器**，也没有显式调用**父类构造器**，系统将会**隐式插入空参数的父类构造器super()作为第一条指令**



JAVA

```
public class B extends A{
        public static void main(String[] args){
                B b = new B(2);
        }
        public B(int i){
                System.out.println(i);
        }
}
class A{
        int i;
        A(int i){
                this.i = i * 2;
        }
}
```

这份代码会报错，因为在`B`的构造器中没有显式调用`A`的构造器，故JVM会隐式调用一个空构造器，但父类中没有空的构造方法，所以报错

构造器链

下面的例子因为会*隐式调用* `super()`，但没有传参数，会报错

```java
public class Apple extends Fruit{
        public class Apple(){} // 会报错，因为构造器没有值传入，且父类没有空构造器可以用`super`调用
}
class Fruit{
        public Fruit(String name){
                System.out.println("Fruit's constructor");
        }
}
```

但以下的例子不会：

JAVA

```java
public class Faculty extends Employee{
        public static void main(String[] args){
                new Faculty();
        }
        public Faculty(){
                // super(); 隐式调用父类构造器
                System.out.println("(4) Faculty's no-arg constructor is invoked");
        }
}
class Employee extends Person{
        public Employee(){
                // super(); 同样隐式调用
                this("(2) Invoke Employee's overloaded constructor");
                System.out.println("(3) Employee's no-arg constructor is invoked");
        }
        public Employee(String s){
                System.out.println(s);
        }
}
class Person{
        public Person(){
                System.out.println("(1) Person's no-arg constructor is invoked");
        }
}
// 打印:
// (1) Person's no-arg constructor is invoked
// (2) Invoke Employee's overloaded constructor
// (3) Employee's no-arg constructor is invoked
// (4) Faculty's no-arg constructor is invoked
```

当执行`new Faculty();`时，会发生以下步骤:

1. `Faculty`的构造方法被调用。
2. 由于`Faculty`继承自`Employee`，因此在`Faculty`的构造方法中，会隐式地调用`Employee`的无参数构造方法。
3. 在`Employee`的无参数构造方法中，首先会隐式地调用`Person`的无参数构造方法（因为`Employee`继承自`Person`）*1*。
4. 接着，会执行`Employee`无参数构造方法中的其他语句，其中，在`Employee`的无参数构造方法中，通过`this`关键字调用`Employee`的带有String参数的构造方法 *2*，然后再打印 *3*。
5. 最后打印`Faculty`构造方法的最后一句 *4*

总结

方法和构造器的重载

- 方法的重载是实现Java语言多态机制的重要手段
  - 同一个方法名对应多个不同的实现　如Employee a = new Manager/Director
  - 引用变量多态是指同一个变量可引用多种类型的对象（子类）
- 方法的重载与方法的覆盖不同
  - 方法的重载不是子类对父类同名方法的重新定义，而是同一类中若干同名方法的重新定义
- 系统仅通过参数列表选择匹配的方法定义，不能定义参数相同，返回值不同的同名方法
- 构造函数的重载：是指在同一个类中存在若干个具有不同参数列表的构造函数
  - 创建该类对象的语句会自动根据给出的实际参数的数目、类型和排列顺序调用相应的构造函数来完成对新对象的初始化操作

示例

```
class SuperC {
   public void methodA(int i){System.out.println("methodA(int) in SuperC");}
   protected void methodB(){System.out.println("methodB() in SuperC");}
   void methodC(){System.out.println("methodC() in SuperC");}
   public void methodD(String s, int i) throws IOException{
      System.out.println("methodD(String, int) in SuperC");
   }
   public int methodE(String s, int i) {
      System.out.println("methodE(String, int) in SuperC"); return 0;
   }
}
class SubC extends SuperC {
   public void methodA(int i){System.out.println("methodA(int) in SubC");}
   public void methodB(){System.out.println("methodB() in SubC");}

   private void methodC(){System.out.println("methodC() in SubC");}
   public void methodD(String s, int i) throws Exception {
      System.out.println("methodD(String, int) in SubC");
   }
   public double methodE(String s, int i){
      System.out.println("methodE(String, int) in SubC"); return 0;
   }
}
```

methodC()权限缩小
methodE()参数相同，返回类型不同，发生重载

```
import java.io.IOException;
public class OverridenTest {
  public static void main(String[] args) {
     SuperC c = new SubC();
     c.methodA(0); c.methodB();
     c.methodC();  c.methodD("hello", 3);
     c.methodE("aaa", 2);
  }
}
```

静态方法

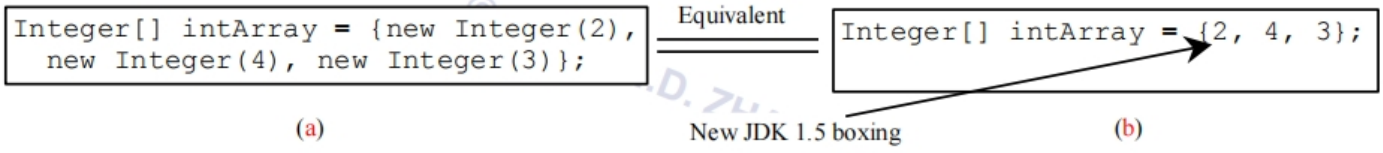
静态方法用**static**修饰，可以用类名访问

```java
Double doubleObject = Double.valueOf("12.4");
Integer integerObject = Integer.valueOf("12");
String stringObject = String.valueOf(12.4);
```

`Double.valueOf()`，`Integer.valueOf()`，和 `String.valueOf()` 都是Java类库中的静态方法。

静态方法*在类加载时就存在*，**不需要创建类的实例**就可以调用。它们通常用于执行与类本身相关的操作，而不是与类的特定实例相关的操作。

- `Double.valueOf("12.4")` 是静态方法，它接收一个字符串参数，并将其转换为一个 `Double` 对象。
- `Integer.valueOf("12")` 也是静态方法，它接收一个字符串参数，并将其转换为一个 `Integer` 对象。
- `String.valueOf(12.4)` 同样是一个静态方法，它将接收的参数转换为字符串。
  这些静态方法都是各自类的一部分，可以**通过类名直接调用**，无需创建类的对象实例。




```
Integer[] intArray = {1, 2, 3};
System.out.println(intArray[0] + intArray[1] + intArray[2]);
```

static 关键字

- `static` 关键字定义与类关联的属性、方法等，而不是与变量引用的对象关联
- 实例变量（instance variables），系统为*每个对象的实例变量分配一块内存*，**不同对象**的实例变量**各不相同**，外部只能通过对象来访问
- 类变量（class variables）用关键字`static`修饰，系统装载类时，分配类变量的内存
  - 生成类的实例对象时，将*共享这块内存*（类变量）
  - 任何一个对象对类变量的修改，都会*影响同类其他对象*对这个变量的访问
  - 父类中的静态方法在子类中重写，*静态绑定*
- `static`不可以修饰类

静态变量

- 类变量没有定义为`private`，则可以通过两种外部访问方式对静态变量进行访问：*通过对象访问、通过类名访问*

关于静态变量的内存分配：

```java
public class Puzzle{
        private int state;
        private static final FLAG;
        public void scramble(){}
        public boolean move(){}
        public static void main(String s[]){
                Puzzle p = new Puzzle();
        }
}
```

在Java中，内存主要分为以下几个部分：方法区（Method Area），堆区（Heap），栈区（Stack）和程序计数器（Program Counter Register）。你给出的代码中，每个部分分别存储在以下区域：

1. `public class Puzzle{}` - 这个类定义本身存储在方法区。方法区是存储类信息，常量，静态变量等非堆内存区域。
2. `private int state;` - 这是一个**实例变量**，它存储在**堆区**。每当创建`Puzzle`类的一个新实例时，Java就会在堆上为这个实例变量分配内存。
3. `private static final FLAG;` - 这是一个**静态变量**，所以它存储在**方法区**。因为它是`final`的，所以它的值在编译时就确定了，并被存储在方法区的常量部分。
4. `public void scramble(){}` 和 `public boolean move(){}` - 这两个方法定义存储在**方法区**。
5. `public static void main(String s[]){ Puzzle p = new Puzzle(); }` 这个`main`方法定义也存储在方法区。在`main`方法中创建的`Puzzle`对象`p`在堆上分配内存。而`String s[]`这个参数在栈上分配内存。

总结起来，`Puzzle`类的定义，它的方法，以及静态变量都存储在方法区。`Puzzle`对象的实例变量存储在堆区。在方法中创建的局部变量和对象的引用存储在栈区。

静态变量的访问

- 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**某个类的类名来访问**它的静态变量

```java
public class Sample1 {
        public static int number; //定义一个静态变量
        public void method() {
                int x = number; //在类的内部直接访问number静态变量
        }
}
public class Sample2 [
        public void method() {
                int x = Sample1.number; //通过Sample1类名来访问number静态变量。
        }
}
```

静态变量的作用之一：编号

eg. 第一个对象创建时编码为1,后面的依次为2,3,…

JAVA

```java
public class Count{
        private int serialNumber;
        public static int counter = 0;  // 定义静态变量
        public Count(){
                counter++; serialNumber= counter;
        }
        public void print(){
                System.out.println(serialNumber);
        }

        public static void main(String s[]){
                Count c1 = new Count(); c1.print();
                //Count.counter++;
                Count c2 = new Count(); c2.print();
        }
}
```

由于类中共享一个静态变量，故其值不会因为实例变量初始化而重新定义为0。且由于构造器中设置`counter++`，故其会随着实例变量不断创造而增加值

类方法

- 类方法使用*关键字static*标记，当Java虚拟机加载类时，就会执行该代码块。用于还没有生成类的实例对象，却要**使用类中定义的方法**的情况，类方法也可以使用两种方法来访问：
  - 类方法尽量使用`ClassName.staticMethod()`来访问
  - 实例方法尽量用`object.instanceMehod()`来访问
  - 读代码的时候一看就知道哪些方法是类方法，哪些方法是实例方法。
- 用类方法可能在没有对象创建前调用，所以类方法除了使用局部变量、类变量、类方法和参数外，其它变量不可访问。
- **类方法不可以使用this**，有可能没创建对象
- 重写的时候子类与父类的同名且参数相同的方法，要么同为同为实例方法
- 调用的时候不是动态确定的。
- `main`方法是`static`方法

JAVA

```java
public class Sample1 {
        public static int add(int x, int y) {return x + y; }
}
public class Sample2 {
        public void method() {
                int result = Sample1.add(1,2); // 不创建类，直接调用
                System.out.println("result= " + result);
        }
}
```

静态初始器

- 静态初始器（Static Initializer）是一个存在于**类中方法外面**的静态块，方法外面只有变量声明和静态初始化器、构造器
- 静态初始器仅仅在类装载的时候（**第一次使用类**的时候）执行一次。不是每次创建对象都要执行的静态初始器**往往用来初始化静态的类属性**

JAVA

```java
public class Count{
        private int serialNumber;
        private static int counter=0;

        static {  // 一开始启动，将counter初始化为10
        counter = 10;
        }

        public Count(){
                counter++; serialNumber= counter;
        }
        public void print(){
                System.out.println(serialNumber);
        }
        public static void main(String s[]){
                Count c1 = new Count();
                c1.print();  // 11
                Count c2 = new Count();
                c2.print();  // 12
        }
}
```

JAVA

```java
public class StaticInitTest {
        public static void main(String[] args) {
                System.out.println("Main code i = " + StaticInitDemo.i);
                System.out.println();
                System.out.println("Main code i = " + StaticInitDemo.i);
                new StaticInitDemo();
        }
}
class StaticInitDemo {
        static int i = 5;
        static {
                System.out.println("Static code i= "+ i++ );
        }
}
// 先执行完构造器，再进行输出
// Staric code i = 5
// Main code i = 5
// Main code i = 6
```

static 代码块

- 类中可以包含静态代码块，它不存在于任何方法体中。Java虚拟机加载类时，会执行这些静态代码块
- 如果类中包含多个静态块，那么Java虚拟机按它们在类中出现的顺序依次执行它们，每个静态代码块**只会被执行一次**
- 运行`Sample`类的`main()`方法时，Java虚拟机首先加载`Sample`类，在加载的过程中依次执行两个静态代码块。Java虚拟机加载`Sample`类后，再执行`main()`方法

JAVA

```
public class Sample{
        static int i = 5;
        static { //第一个静态代码块
                System.out.println(" First Static code i= "+ i++ );
        }
        static { //第二个静态代码块
                System.out.println(" Second Static code i= "+ i++ );
        }
        public static void main(String args[]) {
                Sample s1 = new Sample();
                //Sample s2=new Sample();
                System.out.println("At last, i= "+ i );
        }
}
//  First Static code i= 5
//  Second Static code i= 6
//  At last, i= 7
```

Object类及方法

`Object`类的`toString()`方法可将任何对象转化为字符串，需要时编译器自动转换
`toString()`通常返回类名，重写可根据需要输出提示信息

JAVA

```
public class MyDate{
        private int day, month, year;
        public MyDate(int d, int m, int y){day = d; month = m; year = y;}
        public String toString(){
                return "MyDate:" + year + "--" + month + "--" + day;
        }
        public static void main(String args[]){
                MyDate one = new MyDate(1, 10, 2006);
                System.out.println(one);
        }
}
```

`toString`方法是Java中的一个重要方法，用于返回对象的字符串表示形式。每个Java对象都继承自`Object`类，而`Object`类提供了一个默认的`toString`方法。这个默认方法*通常会返回对象的类名以及哈希码的无符号十六进制表示*。

在上面的代码中，`MyDate`类重写了`toString`方法，以返回日期的特定格式字符串表示。重写`toString`方法使得当你尝试打印`MyDate`对象或将其转换为字符串时，会得到一个更有意义和可读的输出。

具体来说，这个`toString`方法返回一个字符串，格式为`"MyDate:年-月-日"`。例如，对于日期2006年10月1日，它会返回字符串`MyDate:2006--10--1`。

包装类

- Java语言为了提高效率，将八种原始类型**不看作对象**。但对每种类型提供了一个**包装类**，封装一个值
  - **引用类型**可以指向别人，但指向对应变量后，值不可变
  - 注意**封装类为不可变**的（*免疫对象*），一旦初始化后，就无法再改变其中封装的值

```
int pInt=500;
Integer wInt=new Integer(pInt);
int p2 = wInt.intValue();

int x;
String str="123";
x=Integer.valueOf(str).intValue();
x=Integer.parseInt(str);
```

Primitive Data Type	Wrapper Class
boolean	Boolean
byte	Byte
char	Character
short	Short
int	Integer
long	Long
float	Float
double	Double

`intValue()`为`Integer`类方法，作用是返回该变量的数值属性
`valueOf(String)`方法作用是将读入的字符串转化为`Integer`对象

Integer和Double类：构造器，常数，转化方法

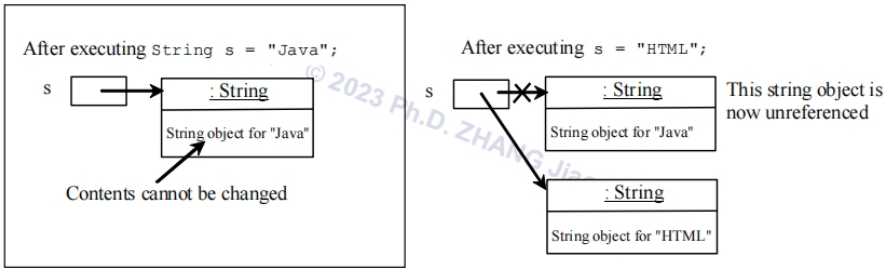
java.lang.Integer
-value: int +MAX_VALUE: int +MIN_VALUE: int
+Integer(value: int) +Integer(s: String) +byteValue(): byte +shortValue(): short +intValue(): int +longVlaue(): long +floatValue(): float +doubleValue():double +compareTo(o: Integer): int +toString(): String +valueOf(s: String): Integer +valueOf(s: String, radix: int): Integer +parseInt(s: String): int +parseInt(s: String, radix: int): int

java.lang.Double
-value: double +MAX_VALUE: double +MIN_VALUE: double
+Double(value: double) +Double(s: String) +byteValue(): byte +shortValue(): short +intValue(): int +longVlaue(): long +floatValue(): float +doubleValue():double +compareTo(o: Double): int +toString(): String +valueOf(s: String): Double +valueOf(s: String, radix: int): Double +parseDouble(s: String): double +parseDouble(s: String, radix: int): double

String类型

String类型变量不可更改（续）

```
String s = "Java";
s = "HTML";
```



更改`String`变量值实际上是*改变指针的指向*

字符串替换与分割

字符串替换和分割

java.lang.String	
+replace(oldChar: char, newChar: char): String	Returns a new string that replaces all matching character in this string with the new character.
+replaceFirst(oldString: String, newString: String): String	Returns a new string that replaces the first matching substring in this string with the new substring.
+replaceAll(oldString: String, newString: String): String	Returns a new string that replace all matching substrings in this string with the new substring.
+split(delimiter: String): String[]	Returns an array of strings consisting of the substrings split by the delimiter.

```
"Welcome".replace('e', 'A') returns a new string, WAlcomA.
"Welcome".replaceFirst("e", "AB") returns a new string, WABlcome.
"Welcome".replace("e", "AB") returns a new string, WABlcomAB.
"Welcome".replace("el", "AB") returns a new string, WABcome.
```

```
String[] tokens = "Java#HTML#Perl".split("#", 0);
for (int i = 0; i < tokens.length; i++)
    System.out.print(tokens[i] + " ");
```

Java HTML Perl

StringBuffer/StringBuilder

`StringBuffer`、`StringBuilder`和`String`在Java中的区别主要有：

1. 可变性：`String`类是**不可改变**的，一旦创建了字符串对象，就无法修改其内容。而`StringBuffer`和`StringBuilder`是**可变**的，可以修改字符串的内容。
2. 线程安全：`String`类和`StringBuffer`类是线程安全的，而`StringBuilder`类是非线程安全的。
3. 性能：在处理字符串时，`StringBuffer`相较于`String`具有优势，因为`StringBuffer`在处理字符串时不会生成新的对象，从内存角度来说，`StringBuffer`更优。

因此，如需**处理大量的字符串操作**，建议使用`StringBuffer`或`StringBuilder`，以提高性能和效率；如果只是简单的字符串操作，且不需要修改字符串内容，可以使用`String`。

Singleton设计范式(单例模式)

- Singleton设计范式(Design Pattern)保证整个程序运行当中仅仅生成一个实例(如太阳、地球等)

实现

- 构造器声明为`private`，外面不可以随意初始创建实例；
- 定义**本类类型的静态属性**，并初始化一个实例；

JAVA

```
public class Moon{
        private int height;
        private static Moon instance = new Moon(38000);  // 定义静态实例instance
        private Moon(int h){height = h;}  // 私有构造器
        public static Moon getMoon(){return instance;}
}  // 访问该私有变量
class Test{
        public static void main(String[] args){
                // Moon m1 = new Moon(40000);
                // 报错，因为构造器设置为private，无法从别的类访问
                Moon m2 = Moon.getMoon();  //
                Moon m3 = Moon.getMoon();
                System.out.println(m2 == m3); // true
        }
}
```

`Moon` 类中定义了一个私有的静态实例 `instance`，并通过 `getMoon()` 方法提供对这个实例的访问。因为 `instance` 是静态的，它会在 `Moon` 类加载时被创建，并且**只有一个**。

当你调用 `Moon.getMoon()` 方法时，它总是返回这个已经创建的静态实例 `instance`，所以 `m2` 和 `m3` 指向的是同一个对象，因此 `m2 == m3` 会返回 `true`。

简单工厂范式

- 定义一个工厂类，它可以*根据参数的不同返回不同类的实例*，被创建的实例通常都具有共同的父类
- 因为在简单工厂模式用于创建实例的方法是**静态的方法**，因此简单工厂模式又被称为静态工厂方法模式，它属于*类创建型*模式（不需要创建类对应的实例即可使用其方法）
  实验中有两道关于此的练习： 实验6.2_姓名的拆解_202211079261_何嘉凯

JAVA

```
class Factory{
        public static Product get(ConcreteProduct type){
                switch (type){
                        case A: return new ConcreteProductA();
                        case B: return new ConcreteProductB();
                        default: return null;
                }
        }
}
```

练习：

课堂小问题(7)

Analyze the following two classes.

```
class First {
  static int a = 3;
}

final class Second extends First {
  void method() {
    System.out.println(a);
  }
}
```

a) Class First compiles, but class Second does not
b) Class Second compiles, but class First does not
c) Neither class compiles
d) Both classes compile, and if method() is invoked, it writes 3 to the standard output
e) Both classes compile, but if method() is invoked, it throws an exception

答案：d

This code is perfectly fine, and subclass Second can access the static variable in its superclass

巨坑！A没有空构造器啊啊啊

课堂小问题(8)

What will happen if you try to compile and execute B's main() method?

```
class A {
  int i;
  A(int i) {    this.i = i * 2;   }
}

class B extends A {
  public static void main(String[] args) {
    B b = new B(2);
  }
  B(int i) {    System.out.println(i);   }
}
```

a) The instance variable i is set to 4
b) The instance variable i is set to 2
c) The instance variable i is set to 0
d) This code will not compile

答案：d

B`s constructor implicitly calls A`s no-args constructor, and such a constructor is not defined.

课堂小问题(10)

```
1. class BaseWidget extends Object{
2.   String name="BaseWidget";
3.   void speak(){System.out.println("I am a "+name);}
4. }
5. class TypeAWidget extends BaseWidget{
6.   TypeAWidget(){name="TypeA";}
7. }
```

答案：b

Which code fragments will compile and execute without error?

a. Object A=new BaseWidget();
A.speak();

b. BaseWidget B=new TypeAWidget();
B.speak();

c. TypeAWidget C=new BaseWidget();
C.speak();

```
BaseWidget sample = new TypeAWidget();
sample.speak();
```

输出 "I am a TypeA"

课堂小问题(3)

以下代码能否编译通过，假如能编译通过，运行时得到什么打印结果？

```
class Test{
  int x = 5;
  static String  s = "abcd";
  public static void method(){
    System.out.println(s + x);
  }
}
```


答案：编译出错，静态method()方法不允许访问实例变量x

课堂小问题(5)

What is written to the standard output as the result of the following statements?

```
Boolean b1 = new Boolean(true);
Boolean b2 = new Boolean(true);
Object obj1 = (Object)b1;
Object obj2 = (Object)b2;
if (obj1 == obj2)
  if (obj1.equals(obj2))
    System.out.println("a");
  else
    System.out.println("b");
else
  if (obj1.equals(obj2))
    System.out.println("c");
  else
    System.out.println("d");
```

A) a B) b C) c D) d

答案：C

even though we`ve cast the Boolean objects to be of type Object, the method invoked still goes to the actual object type, which is Boolean. So, as is the case here, equals() will return true in the case of Boolean objects assigned to the same boolean value.

1. `Boolean b1 = new Boolean(true);`
2. `Boolean b2 = new Boolean(true);`
   这两行代码创建了两个分别代表 `true` 值的 `Boolean` 对象，`b1` 和 `b2`。

3. `Object obj1 = (Object)b1;`

4. `Object obj2 = (Object)b2;`
   这两行代码将 `b1` 和 `b2` 强制转换为 `Object` 类型。但是，这并不改变 `obj1` 和 `obj2` 仍然是指向两个不同 `Boolean` 对象的引用这一事实。

5. `if (obj1 == obj2)`
   这行代码检查 `obj1` 和 `obj2` 是否指向同一个对象。由于 `b1` 和 `b2` 是两个不同的 `Boolean` 对象，因此 `obj1` 和 `obj2` 是不同的引用。因此，这个条件为 `false`。

6. `else if (obj1.equals(obj2))`
   由于第一个 `if` 条件为 `false`，代码检查这个 `else if` 条件。这行代码检查 `obj1` 和 `obj2` 引用的对象是否在值上相等。`obj1` 和 `obj2` 都引用代表 `true` 的 `Boolean` 对象。`Boolean` 的 `equals` 方法会检查两个对象是否代表相同的值。因为它们确实如此，所以这个条件为 `true`。