泛型

本节目标

> 1. JDK1.5新特性简介
> 2. 泛型

1. 新特性(JDK1.5)

从JDK1.0开始，几乎每个版本都会提供新特性。例如在JDK中有以下代表性版本：

> - JDK1.2: 推出了轻量级的界面包：Swing
> - JDK1.5: 推出新程序结构的设计思想。
> - JDK1.8: Lambda表达式、接口定义加强

现在已经接触了新特性：自动拆装箱、switch对String的支持

1.1 可变参数

现在假设说有这样的要求：要求设计一个方法，用于计算任意参数的整数的相加结果。

这个需求在早期只能通过数组的方式来实现。

范例：早期实现方式

```
public class TestDemo {
    public static void main(String[] args) {
        System.out.println(add(new int[]{1}));
        System.out.println(add(new int[]{1,2,3}));
        System.out.println(add(new int[]{1,2,3,4,5,6,7,8}));
    }
    public static int add(int[] data) {
        int result = 0 ;
        for (int i = 0; i < data.length; i++) {
            result += data[i] ;
        }
        return result ;
    }
}
```

这种最初的实现方式本身存在缺陷，现在要求设计的不是数组，而是任意多个参数。从JDK1.5之后追加了可变参数的概念，这个时候方法的定义格式：

```
public [static] [final] 返回值 方法名称([参数类型 参数名称][参数类型 ... 参数名称])
{}
```

比特科技

这个参数上使用的“...”实际上表示一个数组的结构。

范例：方法的可变参数

```
public class TestDemo {
    public static void main(String[] args) {
        System.out.println(add(1,4,5,6)); // 随意传递的内容，随意个数
        System.out.println(add(new int[]{1,2,3})); // 可变参数可以接收数组
        System.out.println(add(new int[]{1,2,3,4,5,6,7,8}));
    }
    public static int add(int ... data) { // 本身还是一个数组
        int result = 0 ;
        for (int i = 0; i < data.length; i++) {
            result += data[i] ;
        }
        return result ;
    }
}
```

现阶段可以通过类库观察到使用。

注意点：如果要传递多类参数，可变参数一定放在最后，并且只能设置一个可变参数

范例：传递多类参数

```
public class TestDemo {
    public static void main(String[] args) {
        System.out.println(add("Hello"));
        System.out.println(add("Hello",1,4,5,6));
        System.out.println(add("Hello",new int[]{1,2,3}));
    }
    public static int add(String msg,int ... data) {
        int result = 0 ;
        for (int i = 0; i < data.length; i++) {
            result += data[i] ;
        }
        return result ;
    }
}
```

以后要想编写一些更好的程序方法，可变参数一定会作为考虑的方法。

1.2 foreach循环

范例：原始数组的输出使用for循环完成。

比特科技

```
package www.bit.java.testdemo;

public class TestDemo {
    public static void main(String[] args) {
        int[] data = new int[] { 1, 2, 3, 4, 5 }; // 原始数组
        for (int i = 0; i < data.length; i++) {
            System.out.println(data[i]); // 通过循环控制索引下标
        }
    }
}
```

从JDK1.5之后对于for循环的使用有了新格式：

```
for(数据类型 临时变量 : 数组(集合)) {
// 循环次数为数组长度，而每一次循环都会顺序取出数组中的一个元素赋值给临时变量
}
```

即在for循环里面无须使用索引来取数据

范例：使用foreach循环

```
package www.bit.java.testdemo;

public class TestDemo {
    public static void main(String[] args) {
        int[] data = new int[] { 1, 2, 3, 4, 5 }; // 原始数组
        for (int i : data) { // 将数组中每个元素赋值给i
            System.out.println(i); // 这种循环避免了角标的问题
        }
    }
}
```

通过此方式可以很好的避免数组越界的问题，但是这种数组的操作只适合简单输出模式。

1.3 静态导入(了解)

定义一个`MyMath`类，这个类提供了`static`方法。

范例：定义`MyMath`类

比特科技

```
package www.bit.java.util;
public class MyMath {
    public static int add(int x, int y) {
        return x + y;
    }
    public static int sub(int x, int y) {
        return x - y;
    }
}
```

该类中所有方法均为静态方法，于是按照最初的使用原则。首先导入`MyMath`类，然后通过`MyMath`类调用所有静态方法。

范例：使用`MyMath`类

```
package www.bit.java.test;
import www.bit.java.util.MyMath;
public class TestDemo {
        public static void main(String[] args) {
                System.out.println(MyMath.add(10, 20));
                System.out.println(MyMath.sub(30, 10));
        }
}
```

从JDK1.5开始，如果类中方法全是static方法，则可以直接把这个类的方法导入进来，这样就好比像在主类中定义的方法那样，可以被主方法直接调用。

范例：静态导入

```
package www.bit.java.test;

import static www.bit.java.util.MyMath.*; // 静态导入

public class TestDemo {
        public static void main(String[] args) {
                System.out.println(add(10, 20));
                System.out.println(sub(30, 10));
        }
}
```

这种代码出现的几率不高，能看懂即可，在自己开发之中不建议使用。

2.泛型

从JDK1.5以后引入了三大常用新特性：泛型、枚举(enum)、注解（Annotation）。其中在JDK1.5中泛型是一件非常重要的实现技术，它可以帮助我们解决程序的参数转换问题。

2.1 问题引出

比特科技

假设需要你定义一个描述坐标的程序类Point，需要提供两个属性x、y。对于这两个属性的内容可能有如下选择：

> 1. x = 10、y = 20；
> 2. x = 10.1、y = 20.1；
> 3. x = 东经80度、y = 北纬20度

那么现在首先要解决的问题就是Point类中的x、y的属性类型问题，此时需要保存的有int、double、String，所以在java中只有一种类型可以保存所有类型：Object型

范例：定义Point类

```
class Point {
        private Object x ;
        private Object y ;
        public Object getX() {
                return x;
        }
        public void setX(Object x) {
                this.x = x;
        }
        public Object getY() {
                return y;
        }
        public void setY(Object y) {
                this.y = y;
        }
}
```

范例：设置整型坐标

```
// 设置数据
Point p = new Point() ;
p.setX(10); // 自动装箱并且向上转型为Object
p.setY(20);
// 取出数据
int x = (Integer) p.getX() ; // 强制向下转型为Integer并且自动拆箱
int y = (Integer) p.getY() ;
System.out.println("x = " +x+",y = "+y);
```

范例：设置字符串

比特科技

```
// 设置数据
Point p = new Point() ;
p.setX("东经80度");
p.setY("北纬20度");
// 取出数据
String x = (String) p.getX() ;
String y = (String) p.getY() ;
System.out.println("x = " +x+",y = "+y);
```

以上代码看起来已经解决问题，但是现在解决问题的关键在于Object，于是问题也就出现在Object上。

范例：观察程序问题

```
// 设置数据
Point p = new Point() ;
p.setX(10.2);
p.setY("北纬20度");
// 取出数据
String x = (String) p.getX() ;
String y = (String) p.getY() ;
System.out.println("x = " +x+",y = "+y);
```

这个时候由于设置方的错误，将坐标内容设置成了double与String，但是接收方不知道，于是在执行时就会出现`ClassCastException`。`ClassCastException`指的是两个没有关系的对象进行强转出现的异常。

这个时候语法不会对其做任何限制，但执行的时候出现了程序错误，所以得出结论：向下转型是不安全的操作，会带来隐患。

2.2 基本使用

泛型指的就是在类定义的时候并不会设置类中的属性或方法中的参数的具体类型，而是在类使用时再进行定义。

如果要想进行这种泛型的操作，就必须做一个类型标记的声明。

范例：泛型类的基本语法

```
class MyClass<T> {
        T value1;
}
```

尖括号`<>`中的 T 被称作是**类型参数**，用于指代任何类型。实际上这个T你可以任意写，但出于规范的目的，Java 还是建议我们用单个大写字母来代表类型参数。常见的如：

- T 代表一般的任何类。
- E 代表 Element 的意思，或者 Exception 异常的意思。
- K 代表 Key 的意思
- V 代表 Value 的意思，通常与 K 一起配合使用。

比特科技

- S 代表 Subtype 的意思，文章后面部分会讲解示意。

如果一个类被 `<T>` 的形式定义，那么它就被称为是泛型类。定义泛型类之后如何使用呢？来看下面的代码:

范例：使用泛型类

```
MyClass<String> myClass1 = new MyClass<String>();
MyClass<Integer> myClass2 = new MyClass<Integer>();
```

注意：泛型只能接受类，所有的基本数据类型必须使用包装类！

这里需要注意的是，泛型类可以接收多个类型参数，如下所示:

范例：泛型类引入多个类型参数以及使用

```
class MyClass<T,E> {
        T value1;
        E value2;
}
public class Test {
        public static void main(String[] args) {
                MyClass<String,Integer> myClass1 = new MyClass<String,Integer>
();
        }
}
```

范例：使用泛型定义Point类

```
 package www.bit.java.test;
 class Point <T> { // T表示参数，是一个占位的标记；如果有多个泛型就继续在后面追加
        private T x ;
        private T y ;
        public T getX() {
                return x;
        }
        public void setX(T x) {
                this.x = x;
        }
        public T getY() {
                return y;
        }
        public void setY(T y) {
                this.y = y;
        }
}

public class TestDemo {
```

比特科技

```
        public static void main(String[] args) {
                // 设置数据
                Point<String> p = new Point<String>() ; // JDK1.5的语法
                p.setX("东经80度");
                p.setY("北纬20度");
                // 取出数据
                String x = p.getX() ; // 避免了向下转型
                String y = p.getY() ;
                System.out.println("x = " +x+",y = "+y);
        }
}
```

注意：*Point p = new Point() ;*

此行语句在`JDK1.7`以后可以这么写：*Point p = new Point<>()*

当开发的程序可以避免向下转型，也就意味着安全隐患被消除了。**尽量不要去使用向下转型**。

泛型的出现彻底改变了向下转型的需求。引入泛型后，如果明确设置了类型，则为设置类型；如果没有设置类型，则默认为Object类型。

2.3 泛型方法

泛型不仅可以用于定义类，还可以单独来定义方法。如下所示:

范例：泛型方法定义

```
class MyClass{
        public <T> void testMethod(T t) {
                System.out.println(t);
        }
}
```

泛型方法与泛型类稍有不同的地方是，类型参数也就是尖括号那一部分是写在返回值前面的。`<T>` **中的 T 被称为类型参数，而方法中的 T 被称为参数化类型，它不是运行时真正的参数。**

当然，声明的类型参数，其实也是可以当作返回值的类型的。

范例：使用类型参数做返回值的泛型方法

```
class MyClass{
        public <T> T testMethod(T t) {
                return t;
        }
}
```

泛型方法与泛型类可以共存，如下所示:

范例：泛型方法与泛型类共存

比特科技

```
class MyClass<T>{
        public void testMethod1(T t) {
                System.out.println(t);
        }
        public <T> T testMethod2(T t) {
                return t;
        }
}
public class Test {
        public static void main(String[] args) {
                MyClass<String> myClass = new MyClass<>();
                myClass.testMethod1("hello 泛型类");
                Integer i = myClass.testMethod2(100);
                System.out.println(i);
        }
}
```

上面代码中，MyClass`<T>`是泛型类，testMethod1 是泛型类中的普通方法，而 testMethod2 是一个泛型方法。而泛型类中的类型参数与泛型方法中的类型参数是没有相应的联系的，**泛型方法始终以自己定义的类型参数为准**。

泛型类的实际类型参数是 String，而传递给泛型方法的类型参数是 Integer，两者不相干。

但是，为了避免混淆，如果在一个泛型类中存在泛型方法，那么两者的类型参数最好不要同名。比如，MyClass`<T>`代码可以更改为这样

```
class MyClass<T>{
        public void testMethod1(T t) {
                System.out.println(t);
        }
        public <E> E testMethod2(E e) {
                return e;
        }
}
```

2.4 通配符(重点)

在程序类中追加了泛型的定义后，避免了`ClassCastException`的问题，但是又会产生新的情况：参数的统一问题。

范例：观察程序

```
package www.bit.java.test;

class Message<T> {
        private T message ;

        public T getMessage() {
```

```
                return message;
        }

        public void setMessage(T message) {
                this.message = message;
        }

}

public class TestDemo {
        public static void main(String[] args) {
                Message<String> message = new Message() ;
                message.setMessage("比特科技欢迎您");
                fun(message);
        }
        public static void fun(Message<String> temp){
                System.out.println(temp.getMessage());
        }
}
```

以上程序会带来新的问题，如果现在泛型的类型设置的不是String，而是Integer.

```
public class TestDemo {
        public static void main(String[] args) {
                Message<Integer> message = new Message() ;
                message.setMessage(99);
                fun(message); // 出现错误，只能接收String
        }
        public static void fun(Message<String> temp){
                System.out.println(temp.getMessage());
        }
}
```

我们需要的解决方案：可以接收所有的泛型类型，但是又不能够让用户随意修改。这种情况就需要使用通配符"?"来处理

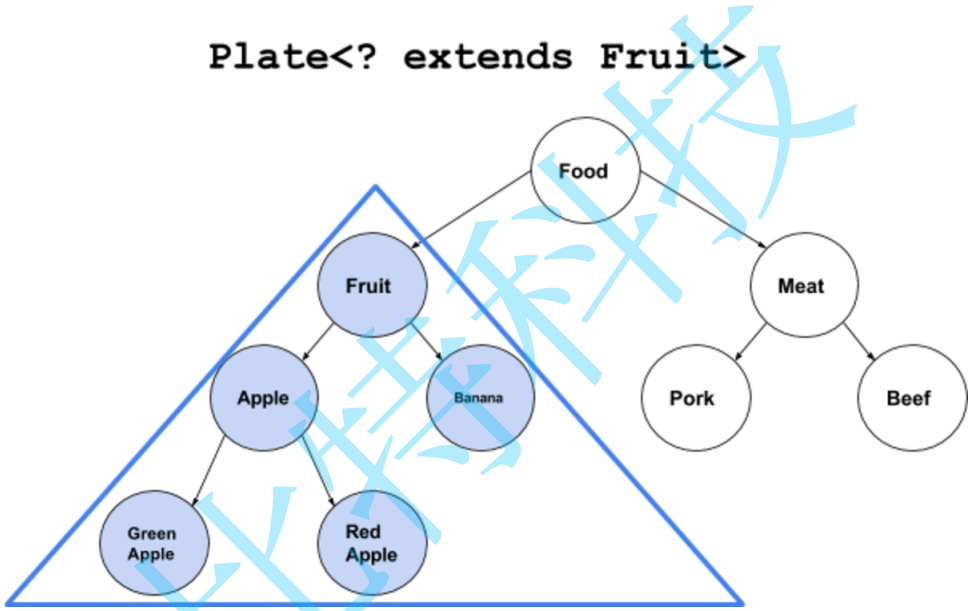
范例：使用通配符

比特科技

```
public class TestDemo {
        public static void main(String[] args) {
                Message<Integer> message = new Message() ;
                message.setMessage(55);
                fun(message);
        }
        // 此时使用通配符"?"描述的是它可以接收任意类型，但是由于不确定类型，所以无法修改
        public static void fun(Message<?> temp){
         //temp.setMessage(100); 无法修改!
                System.out.println(temp.getMessage());
        }
}
```

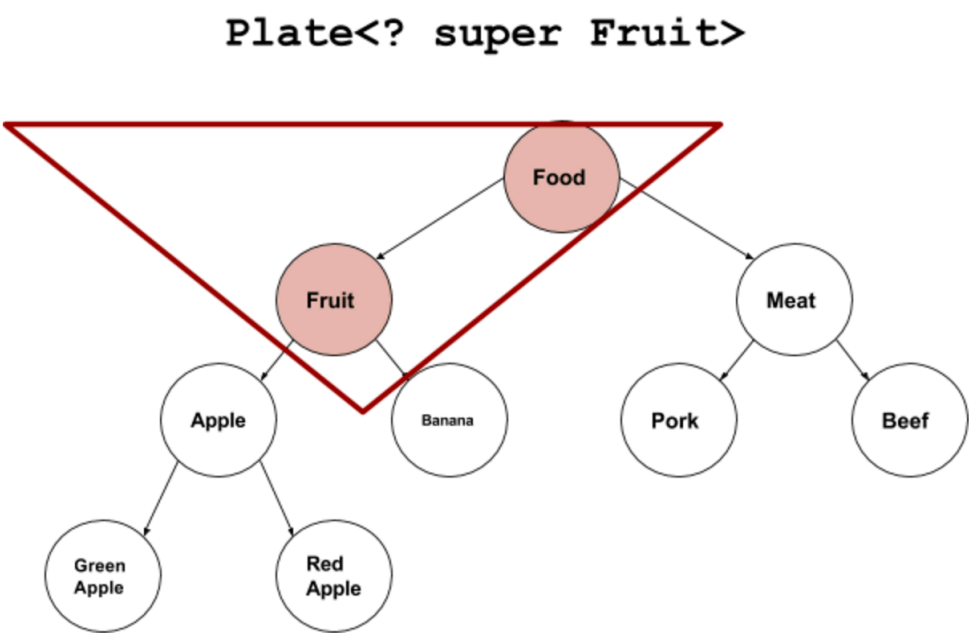
在"?"的基础上又产生了两个子通配符：

- ? extends 类：设置泛型上限：



例如：? extends Number，表示只能够设置Number或其子类，例如：Integer、Double等；

- ? super 类：设置泛型下限：



例如：? super String，表示只能够设置String及其父类Object。

范例：观察泛型上限

```
package www.bit.java.test;

class Message<T extends Number> { // 设置泛型上限
        private T message ;

        public T getMessage() {
                return message;
        }

        public void setMessage(T message) {
                this.message = message;
        }

}

public class TestDemo {
        public static void main(String[] args) {
                Message<Integer> message = new Message() ;
                message.setMessage(55);
                fun(message);
        }
        // 此时使用通配符"?"描述的是它可以接收任意类型，但是由于不确定类型，所以无法修改
        public static void fun(Message<? extends Number> temp){
        //temp.setMessage(100); 仍然无法修改!
                System.out.println(temp.getMessage());
        }
}
```

比特科技

范例：设置泛型下限

```
package www.bit.java.test;
class Message<T> {
        private T message ;
        public T getMessage() {
                return message;
        }
        public void setMessage(T message) {
                this.message = message;
        }
}
public class TestDemo {
        public static void main(String[] args) {
                Message<String> message = new Message() ;
                message.setMessage("Hello World");
                fun(message);
        }
        public static void fun(Message<? super String> temp){
         // 此时可以修改！！
         temp.setMessage("bit!");
                System.out.println(temp.getMessage());
        }
}
```

注意：上限可以用在声明，不能修改；而下限只能用在方法参数，可以修改内容！

以上概念要求大家理解，并不要求立刻使用。

2.5 泛型接口

泛型除了可以定义在类中，也可以定义在接口里面，这种情况我们称之为泛型接口。

范例：定义一个泛型接口

```
interface IMessage<T> { // 在接口上定义了泛型
        public void print(T t) ;
}
```

对于这个接口的实现子类有两种做法

范例：在子类定义时继续使用泛型

```
package www.bit.java.test;

interface IMessage<T> { // 在接口上定义了泛型
        public void print(T t) ;
```

比特科技

```
}
class MessageImpl<T> implements IMessage<T> {
        @Override
        public void print(T t) {
                System.out.println(t);
        }

}
public class TestDemo {
        public static void main(String[] args) {
                IMessage<String> msg = new MessageImpl() ;
                msg.print("Hello World");
        }
}
```

范例：在子类实现接口的时候明确给出具体类型

```
package www.bit.java.test;

interface IMessage<T> { // 在接口上定义了泛型
        public void print(T t) ;
}
class MessageImpl implements IMessage<String> {
        @Override
        public void print(String t) {
                System.out.println(t);
        }

}
public class TestDemo {
        public static void main(String[] args) {
                IMessage<String> msg = new MessageImpl() ;
                msg.print("Hello World");
        }
}
```

以后我们编写的程序一定会使用泛型接口，要求大家一定要掌握。

2.6 类型擦除

泛型是 Java 1.5 版本才引进的概念，在这之前是没有泛型的概念的，但显然，泛型代码能够很好地和之前版本的代码很好地兼容。

这是因为，**泛型信息只存在于代码编译阶段，在进入 JVM 之前，与泛型相关的信息会被擦除掉，专业术语叫做类型擦除。**

通俗地讲，泛型类和普通类在 java 虚拟机内是没有什么特别的地方。

比特科技

来看下面代码:

```
class MyClass<T>{
        private T message;
        public T getMessage() {
                return message;
        }
        public void setMessage(T message) {
                this.message = message;
        }
        public void testMethod1(T t) {
                System.out.println(t);
        }
}
public class Test {
        public static void main(String[] args) {
                MyClass<String> myClass1 = new MyClass<>();
                MyClass<Integer> myClass2 = new MyClass<>();
                System.out.println(myClass1.getClass() ==
myClass2.getClass());
        }

}
```

打印的结果为 true 是因为 `MyClass<String>` 和 `MyClass<Integer>` 在 `JVM` 中的 Class 都是 `MyClass.class`。

范例：观察类型擦除

```
import java.lang.reflect.Field;

class MyClass<T,E>{
        private T message;
        private E text;


        public E getText() {
                return text;
        }
        public void setText(E text) {
                this.text = text;
        }
        public T getMessage() {
                return message;
        }
        public void setMessage(T message) {
                this.message = message;
        }
        public void testMethod1(T t) {
```

比特科技

```
                System.out.println(t);
        }
}
public class Test {
        public static void main(String[] args) {
                MyClass<String,Integer> myClass1 = new MyClass<>();
                Class cls = myClass1.getClass();
                Field[] fields = cls.getDeclaredFields();
                for (Field field : fields) {
                        System.out.println(field.getType());
                }
        }
}
```

在泛型类被类型擦除的时候，之前泛型类中的类型参数部分如果没有指定上限，如 `<T>` 则会被转译成普通的 Object 类型，如果指定了上限如 `<T extends String>` 则类型参数就被替换成类型上限。

比特科技