**day07 java & MyeclipseMyEclipse介绍**

它是Eclipse的插件！

* 选择工作空间！
* 配置jdk
* 配置工作空间的编码：从此之后你创建的任何项目中的文本文件默认编码都是utf-8；
* 字体的配置：

###### 1　debug断点调试

* 设置断点；
* 测试跳入（F5）；
* 测试跳过（F6）；
* 测试跳出（F7）；
* 添加断点；
* 测试进入下一断点；
* 测试返回当前方法栈的头部（Drop To Frame）；
* 清除断点；
* 清除表达式；

注意，停止程序！

###### 2　常用快捷键

　　必须的：

* Alt + /（内容助理）：补全；
* Ctrl + 1（快速定位）：出错时定位错误，与点击“红X”效果一样；
* Ctrl + Shift + O：导包；
* Ctrl + Shift + F：格式化代码块；

不是必须的（自己读，不讲）：

* Ctrl + Shift + T：查看源代码；
* Ctrl + 点击源代码：查看源代码；
* F3：查看选中类的源代码；
* Alt + 左键：查看源代码时的“原路返回”；
* Ctrl + Shift + X：把小写修改为大写；
* Ctrl + Shift + Y：把小写修改为小写；
* Ctrl + Alt + 下键：复制当前行；
* Ctrl + /：添加或撤销行注释；
* Ctrl + Shift + /：对选中代码添加段注释；
* Ctrl + Shift + \：撤销当前段注释；
* Alt + 上键：向上移动当前行；
* Alt + 下键：向上移动当前行；
* Ctrl + D：删除当前行；

## JUnit (单元测试)

###### 1　JUnit的概述

当我们写完了一个类后，总是要执行一下类中的方法，查看是否符合我们的意图，这就是单元测试了。而Junit就是单元测试工具。

* 导包：导入Junit4或以上版本；
* 编写一个类：Person，它就是要被测试的类；
* 编写测试类：PersonTest，给出测试方法，在测试方法上使用@Test注解；
* 执行测试方法。
* 测试方法的作用：可以用来在一个类中给出多个测试方法，相当于一个类中有多个main方法的作用！

Person

|  |
| --- |
| **package** cn.itcast;  **public** **class** Person {  **public** **void** run() {  System.*out*.println("run");  }  **public** **void** eat() {  System.*out*.println("eat");  }  } |

包资源管理器🡪选中Person类🡪右键🡪new🡪JUnit TestCase🡪修改包名为junit.test🡪下一步🡪选中要测试的方法。

PersonTest

|  |
| --- |
| **package** junit.test;  **import** org.junit.Test;  **import** cn.itcast.Person;  **public** **class** PersonTest {  @Test  **public** **void** testRun() {  Person person = **new** Person();  person.run();  }  @Test  **public** **void** testEat() {  Person person = **new** Person();  person.eat();  }  } |

选中某个测试方法，鼠标右键🡪Run as🡪Junit Test，即执行测试方法。

**@Test注解的作用是指定方法为测试方法，测试方法必须是public、void、无参的！！！**

###### 2　@Before和@After(了解)

如果你需要某个方法在每个测试方法之前先执行，那么你需要写一个方法，然后使用@Before来标记这个方法。例如在testRun()和testEat()方法之前需要创建一个Person对象。

PersonTest

|  |
| --- |
| **package** junit.test;  **import** org.junit.Before;  **import** org.junit.Test;  **import** cn.itcast.Person;  **public** **class** PersonTest {  **private** Person person;  @Before  **public** **void** setUp() {  person = **new** Person();  }  @Test  **public** **void** testRun() {  person.run();  }  @Test  **public** **void** testEat() {  person.eat();  }  } |

@After注解标记的方法会在每个执行方法之后执行

@Before和@After标记的方法必须是public、void、无参。

###### 3　@BeforeClass和@AfterClass(了解)

@BeforeClass注解标记的方法会在所有测试方法之前执行，只执行一次！它与@Before不同，@Before标记的方法会在每个测试的方法之前都执行，而@BeforeClass标记的方法会在所有测试方法之前执行一次。

@AfterClass会在所有测试方法执行完之后执行！

@BeforeClass和@AfterClass标记的方法必须是public、static、void、无参的。

## JDK5.0新特性

###### 自动拆装箱

就是8种基本类型与其对应包装器类型可以直接互换！

注意：基本类型的数组，与包装器类型的数组不能互换！

自动拆装箱是JDK5.0的新特性之一，这一特性可以使基本类型，与之对应的包装器类型之间直接转换，例如int的包装器类型是Integer！在JDK5.0之后，你甚至可以把int当作成Integer来使用，把Integer当成int来使用。当然，这不是100%的！

1.1　自动拆装箱概述

在JDK5.0之后，Java允许把基本类型与其对应的包装器类型之间自动相互转换。

* 自动装箱：Integer i = 100，把int类型直接赋值给Integer类型；
* 自动拆装：int a = new Integer(100)，把Integer类型直接赋值给int类型。

1.2　自动拆装箱原理

其实自动拆装箱是由编译器完成的！我们写的代码，再由编译器“二次加工”，然后再编译成.class文件！那么编译器是怎么“二次加工”的呢？

* Integer i = 100：编译器加工为：Integer i = Integer.valueOf(100)；
* int a = i：编译器加载为：int a = i.intValue()；

这也说明一个道理：JVM并不知道什么是自动拆装箱，JVM还是原来的JVM（JDK1.4之前），只是编译器在JDK5.0时“强大”了！

1.3　自动拆装箱演变

大家来看看下面代码：

|  |
| --- |
| Integer i = 100;//这是自动装箱  Object o = i;//这是身上转型 |

上面代码是没有问题的，我们是否可以修改上面代码为：

|  |
| --- |
| Object o = 100; |

ok，这是可以的！通过编译器的处理后上面代码为：

|  |
| --- |
| Object o = Integer.valueOf(100); |

在来看下面代码：

|  |
| --- |
| Object o = Integer.valueOf(100);  int a = o;//编译失败！ |

上面代码是不行的，因为o不是Integer类型，不能自动拆箱，你需要先把o转换成Integer类型后，才能赋值给int类型。

|  |
| --- |
| Object o = Integer.valueOf(100);  int a = (Integer)o; |

1.4　自动拆装箱的误区

来看下面代码：

|  |
| --- |
| int[] intArr = {1,2,3};  Integer[] integerArr = intArr;//编译失败 |

上面代码无论编译通过！因为int[]是引用类型，而不是基本类型。而自动拆装箱只是在基本类型与其对应的包装器类型之间进行转换！也就是说，int[]和Integer[]是两种不同的引用类型。

你可能会认为，把上面代码修改下面的样子即可，这说明需要还是没有理解这一概念。把int[]转换成Integer[]，等同与把Integer转换成String类型一样，是不行的！所以下面代码还是编译失败。

|  |
| --- |
| int[] intArr = {1,2,3};  Integer[] integerArr = (Integer[])intArr; |

　　如果可以把int当成Integer来对待，那么是否可以：int a = null？当然是不行的，所以希望学员们对自动拆装箱正确理解，它不是万能的。

1.5　变态小题目

来看下面代码：

|  |
| --- |
| Integer i1 = 100;  Integer i2 = 100;  boolean b1 = i1 == i2;//结果为true  Integer i3 = 200;  Integer i4 = 200;  boolean b2 = i3 == i4;//结果为false |

你可能对上面代码的结果感到费解，那么我们来打开这个疑团。第一步，我们先把上面代码通过编译器的“二次加工”处理一下：

|  |
| --- |
| Integer i1 = Integer.valueOf(100);  Integer i2 = Integer.valueOf(100);  boolean b1 = i1 == i2;//结果为true  Integer i3 = Integer.valueOf(200);  Integer i4 = Integer.valueOf(200);  boolean b2 = i3 == i4;//结果为false |

这时你应该可以看到，疑团在Integer.valueOf()方法身上。传递给这个方法100时，它返回的Integer对象是同一个对象，而传递给这个方法200时，返回的却是不同的对象。这是我们需要打开Integer的源码（这里就不粘贴Integer的源代码了），查看它的valueOf()方法内容。

Integer类的内部缓存了-128~127之间的256个Integer对象，如果valueOf()方法需要把这个范围之内的整数转换成Integer对象时，valueOf()方法不会去new对象，而是从缓存中直接获取，这就会导致valueOf(100)两次，都是从缓存中获取的同一个Integer对象！如果valueOf()方法收到的参数不在缓存范围之内，那么valueOf()方法会new一个新对象！这就是为什么Integer.valueOf(200)两次返回的对象不同的原因了。

###### 2　可变参数

可变参数就是一个方法可以接收任意多个参数！例如：fun()、fun(1)、fun(1,1)、fun(1,1,1)。你可能认为这是方法重载，但这不是重载，你想想重载能重载多少个方法，而fun()方法是可以传递任何个数的参数，你能重载这么多个方法么？

2.1　定义可变参数方法

public void fun(int… arr) {}

上面方法fun()的参数类型为int…，其中“…”不是省略号，而是定义参数类型的方式。参数arr就是可变参数类型。你可以把上面代码理解为：public void fun(int[] arr)。

|  |
| --- |
| **public** **int** sum1(**int**[] arr) {  **int** sum = 0;  **for**(**int** i = 0; i < arr.length; i++) {  sum += arr[i];  }  **return** sum;  } |
| **public** **int** sum2(**int**... arr) {  **int** sum = 0;  **for**(**int** i = 0; i < arr.length; i++) {  sum += arr[i];  }  **return** sum;  } |

你可能会想，“int[]”和“int…”没有什么不同，只是“int…”是一种新的定义数组形参的方式罢了。那么我应该恭喜你了！没错，这么理解就对了！但要注意，只有在方法的形参中可以使用int…来代替int[]。

2.2　调用带有可变参数的方法

sum1()和sum2()两个方法的调用：

|  |
| --- |
| sum1(new int[]{1,2,3});  sum2(new int[]{1,2,3}); |

这看起来没什么区别！但是对于sum2还有另一种调用方式：

|  |
| --- |
| sum2();  sum2(1);  sum2(1,2);  sum2(1,2,3); |

　　这看起来好像是使用任意多个参数来调用sum2()方法，这就是调用带有可变参数方法的好处了。

2.3　编译器“二次加工”

编译器对sum2方法定义的“二次加工”结果为：

|  |
| --- |
| **public** **int** sum2(**int**[] arr) {  **int** sum = 0;  **for**(**int** i = 0; i < arr.length; i++) {  sum += arr[i];  }  **return** sum;  } |

即把“int…”修改为“int[]”类型。

编译器对sum2方法调用的二次加载结果为：

|  |
| --- |
| sum2(**new** **int**[0]);  sum2(**new** **int**[] {1});  sum2(**new** **int**[] {1, 2});  sum2(**new** **int**[] {1, 2, 3}); |

结论：可变参数其实就是数组类型，只不过在调用方法时方便一些，由编译器帮我们把多个实参放到一个数组中传递给形参。

2.4　可变参数方法的限制

* 一个方法最多只能有一个可变参数；
* 可变参数必须是方法的最后一个参数。

###### 3　增强for循环

增强for循环是for的一种新用法！用来循环遍历**数组**和**集合**。

3.1　增强for的语法

for(元素类型 e : 数组或集合对象) {

}

例如：

int[] arr = {1,2,3};

for(int i : arr) {

System.out.println(i);

}

　　增强for的冒号左边是定义变量，右边必须是数组或集合类型。例如上例中循环遍历的主浊arr这个int数组，增强for内部会依次把arr中的元素赋给变量i。

3.2　增强for的优缺点

* 只能从头到尾的遍历数组或集合，而不能只遍历部分；
* 在遍历List或数组时，不能获取当前元素下标；
* 增强for使用便简单，这是它唯一的优点了；
* 增强for比使用迭代器方便一点！

3.3　增强for原理

任何实现了Iterable接口的类，都有返回Iterator的方法。

其实增强for底层是迭代器，任何实现了Iterable接口的类都可以使用增强for来遍历！

其实增强for内部是使用迭代器完成的！也就是说，任何实现了Iterable接口的对象都可以被增强for循环遍历！这也是为什么增强for可以循环遍历集合的原因（Collection是Iterable的子接口）。

但要注意，Map并没有实现Iterable接口，所以你不能直接使用增强for来遍历它！

|  |
| --- |
| Map<String, String> map = **new** HashMap<String,String>();  map.put("1", "one");  map.put("2", "two");  map.put("3", "three");    **for**(String key : map.keySet()) {  String value = map.get(key);  System.*out*.println(key + "=" + value);  } |

## 泛型

具有一个或多个**类型变量**的类称之为泛型类！

###### 1　泛型概述

泛型是JDK5.0新特性，它主要应用在集合类上。有了泛型之后，集合类与数组就越来越像了。例如：Object[] objs = new Object[10]，可以用来存储任何类型的对象。String[] strs = new String[10]只能用来存储String类型的对象。

ArrayList list = new ArrayList()，可以用来存储任何类型的对象。ArrayList<String> list = new ArrayList<String>()只有用来存储String类型的对象。

1.1　理解泛型类

泛型类具有一到多个泛型变量，在创建泛型类对象时，需要为泛型变量指定值。泛型变量只能赋值为引用类型，而不能是基本类型。例如ArrayList类中有一个泛型变量E，在创建ArrayList类的对象时需要为E这个泛型变量指定值。

list<String> list = new ArrayList<String>();，

其中String就是给List的泛型变量E赋值了。查阅ArrayList的API你会知道，泛型变量E出现在很多方法中：

boolean add(E e)

E get(int index)

因为我们在创建list对象时给泛型类型赋值为String，所以对于list对象而言，所有API中的E都会被String替换。

boolean add(String e)

String get(int index)

也就是说，在使用list.add()时，只能传递String类型的参数，而list.get()方法返回的一定是String类型。

list.add(“hello”);

String s = list.get(0);

1.2　使用泛型对象

创建泛型对象时，引用和new两端的泛型类型需要一致，例如上面的引用是List<String>，而new一端是new ArrayList<String>，两端都是String类型！如果不一致就会出错：

List<Object> list = new ArrayList<String>();//编译失败！

Collection的addAll()方法中使用了通配符（这个概念在下一个基础加强中讲解），所以使用起来比较方便：

|  |
| --- |
| List<Object> list = **new** ArrayList<Object>();  List<String> list1 = **new** ArrayList<String>();  List<Integer> list2 = **new** ArrayList<Integer>();  list.addAll(list1);  list.addAll(list2); |

###### 2　泛型的好处

* 将运行期遇到的问题转移到了编译期；
* 泛型不可能去除所有类型转换，但可以减少了类型转换操作；
* 在循环遍历集合类时，方便了很多；
* 一定程度上提高了安全性。

###### 3　自定义泛型类的语法

3.1　自定义泛型类的语法

public class 类型<一到多个泛型变量的声明> {…}

例如：

public class A<T> {...}

A类中有一个泛型变量的声明（或称之为定义）。这就相当于创建了一个变量一样，然后就可以在类内使用它了。

3.2　泛型类内使用泛型变量

声明的泛型变量可以在类内使用，例如创建实例变量时使用泛型变量指定类型，可以在实例方法中指定参数类型或返回值类型，**但不能在static变量或static方法中使用泛型变量**。

public class A<T> {

private T bean;

public T getBean() {

return bean;

}

public void setBean(T bean) {

this.bean = bean;

}

}

###### 4　继承（或实现）泛型类（或接口）

在Java API中有很多类和接口是泛型的，如果我们需要继承或实现它们时，需要为父类或接口中的泛型变量指定值，例如Comparable就是一个泛型接口。

Comparable

|  |
| --- |
| **public** **interface** Comparable<T> {  **public** **int** compareTo(T o);  } |

该接口中有一个泛型变量T，在实现这个接口时需要为其指定值。一定为其指定值后，所有的T将使用指定的值来替换。

MyComparable

|  |
| --- |
| **public** **class** MyComparable **implements** Comparable<String> {  **public** **int** compareTo(String o) {  **return** 0;  }  } |

在继承（或实现）泛型类（或接口）时还有一种选择，如果当前类是泛型类，也可以把自己的泛型变量赋给父类（或接口）。

MyComparable2

|  |
| --- |
| **public** **class** MyComparable2<X>  **implements** Comparable<X> {  **public** **int** compareTo(X o) {  **return** 0;  }  } |
| MyComparable2<Integer> c = **new** MyComparable2<Integer>(); |

###### 5　泛型方法定义

不只可以创建泛型类，还可以创建泛型方法。可能你会有个误区，下面的方法不是泛型方法：

|  |
| --- |
| class A<T> {  public void fun(T t) {…}  } |

上面fun()方法是泛型类中的方法，它不是泛型方法。

泛型方法是可以自己创建泛型变量，泛型方法中创建的泛型变量只能在本方法内使用。泛型方法可以是实例方法，也可以是静态方法。

public <T> T get(T[] ts, int index) {

return ts[index];

}

我们必须要区别开什么是创建泛型变量，什么是使用泛型变量。其中<T>是创建泛型变量，它必须在返回值前面给出。

泛型方法中的泛型变量只有两个可以使用的点：返回值和参数，而且所有有意义的泛型方法中都会在返回值和参数两个位置上使用泛型变量，但语法上没有强制的要求。

调用泛型方法，通常无需为泛型变量直接赋值，而是通过传递的参数类型间接为泛型变量赋值，例如：

String[] strs = {“hello”, “world”};

String s = get(strs, 0);

　　因为给T[] ts参数赋值为strs，而strs的变量为String[]，所以等同与给get方法的泛型变量赋值为String类型。所以返回值类型T为String类型。

## 泛型擦除

泛型是编译期状态，所有的泛型会被编译器擦除，变成了Object类型！

编译器会在适当的位置上给你添加强转！

## 枚举（鸡肋）

###### 1　什么是枚举类型

　　我们学习过**单例模式**，即一个类只有一个实例。而枚举其实就是**多例**，一个类有多个实例，但实例的个数不是无穷的，是有限个数的。例如word文档的对齐方式有几种：左对齐、居中对齐、右对齐。开车的方向有几种：前、后、左、右！

　　我们称呼枚举类中实例为枚举项！一般一个枚举类的枚举项的个数不应该太多，如果一个枚举类有30个枚举项就太多了！

###### 2　定义枚举类型

定义枚举类型需要使用enum关键字，例如：

|  |
| --- |
| public enum Direction {  FRONT, BEHIND, LEFT, RIGHT;  } |
| Direction d = Direction.FRONT; |

注意，定义枚举类的关键字是enum，而不是Enum，所有关键字都是小写的！

其中FRONT、BEHIND、LEFT、RIGHT都是枚举项，它们都是本类的实例，本类一共就只有四个实例对象。

在定义枚举项时，多个枚举项之间使用逗号分隔，最后一个枚举项后需要给出分号！但如果枚举类中只有枚举项（没有构造器、方法、实例变量），那么可以省略分号！**建议不要省略分号！**

不能使用new来创建枚举类的对象，因为枚举类中的实例就是类中的枚举项，所以在类外只能使用类名.枚举项。

###### 3　枚举与switch

枚举类型可以在switch中使用

|  |
| --- |
| Direction d = Direction.*FRONT*;  **switch**(d) {  **case** *FRONT*: System.*out*.println("前面");**break**;  **case** *BEHIND*:System.*out*.println("后面");**break**;  **case** *LEFT*: System.*out*.println("左面");**break**;  **case** *RIGHT*: System.*out*.println("右面");**break**;  **default**:System.*out*.println("错误的方向");  }  Direction d1 = d;  System.*out*.println(d1); |

　　注意，在switch中，不能使用枚举类名称，例如：“case Direction.FRONT：”这是错误的，因为编译器会根据switch中d的类型来判定每个枚举类型，在case中必须直接给出与d相同类型的枚举选项，而不能再有类型。

###### 4　所有枚举类都是Enum的子类

所有枚举类都默认是Enum类的子类，无需我们使用extends来继承。这说明Enum中的方法所有枚举类都拥有。

* int compareTo(E e)：比较两个枚举常量谁大谁小，其实比较的就是枚举常量在枚举类中声明的顺序，例如FRONT的下标为0，BEHIND下标为1，那么FRONT小于BEHIND；
* boolean equals(Object o)：比较两个枚举常量是否相等；
* int hashCode()：返回枚举常量的hashCode；
* String name()：返回枚举常量的名字；
* int ordinal()：返回枚举常量在枚举类中声明的序号，第一个枚举常量序号为0；
* String toString()：把枚举常量转换成字符串；
* static T valueOf(Class enumType, String name)：把字符串转换成枚举常量。

###### 5　枚举类的构造器

　　枚举类也可以有构造器，**构造器默认都是private修饰，而且只能是private**。因为枚举类的实例不能让外界来创建！

|  |
| --- |
| **enum** Direction {  *FRONT*, *BEHIND*, *LEFT*, *RIGHT*;    Direction() {  System.*out*.println("hello");  }  } |

其实创建枚举项就等同于调用本类的无参构造器，所以FRONT、BEHIND、LEFT、RIGHT四个枚举项等同于调用了四次无参构造器，所以你会看到四个hello输出。

###### 6　枚举类可以有成员

其实枚举类和正常的类一样，可以有实例变量，实例方法，静态方法等等，只不过它的实例个数是有限的，不能再创建实例而已。

|  |
| --- |
| **enum** Direction {  *FRONT*("front"), *BEHIND*("behind"), *LEFT*("left"), *RIGHT*("right");    **private** String name;    Direction(String name) {  **this**.name = name;  }    **public** String getName() {  **return** name;  }  } |
| Direction d = Direction.*FRONT*;  System.*out*.println(d.getName()); |

因为Direction类只有唯一的构造器，并且是有参的构造器，所以在创建枚举项时，必须为构造器赋值：FRONT(“front”)，其中”front”就是传递给构造器的参数。你不要鄙视这种语法，你应该做的是接受这种语法！

Direction类中还有一个实例域：String name，我们在构造器中为其赋值，而且本类还提供了getName()这个实例方法，它会返回name的值。

###### 7　枚举类中还可以有抽象方法（了解）

　　还可以在枚举类中给出抽象方法，然后在创建每个枚举项时使用“特殊”的语法来重复抽象方法。所谓“特殊”语法就是匿名内部类！也就是说每个枚举项都是一个匿名类的子类对象！

通常fun()方法应该定义为抽象的方法，因为每个枚举常量都会去重写它。

你无法把Direction声明为抽象类，但需要声明fun()方法为抽象方法。

|  |
| --- |
| **enum** Direction {  *FRONT*() {  **public** **void** fun() {  System.*out*.println("FROND：重写了fun()方法");  }  },  *BEHIND*() {  **public** **void** fun() {  System.*out*.println("BEHIND：重写了fun()方法");  }  },  *LEFT*() {  **public** **void** fun() {  System.*out*.println("LEFT：重写了fun()方法");  }  },  *RIGHT*() {  **public** **void** fun() {  System.*out*.println("RIGHT：重写了fun()方法");  }  };    **public** **abstract** **void** fun();  } |

###### 8　每个枚举类都有两个特殊方法

每个枚举类都有两个不用声明就可以调用的static方法，而且这两个方法不是父类中的方法。这又是枚举类特殊的地方，下面是Direction类的特殊方法。

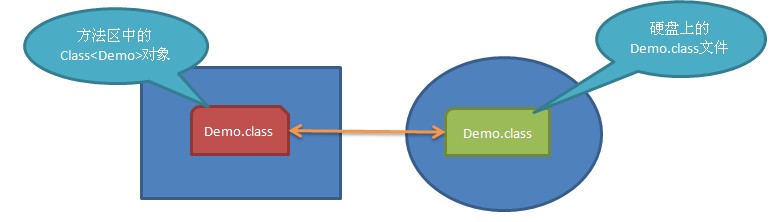
* static Direction[] values()：返回本类所有枚举常量；
* static Direction valueOf(String name)：通过枚举常量的名字返回Direction常量，注意，这个方法与Enum类中的valueOf()方法的参数个数不同。

## 反射概述

###### 1　反射的概述

1.1　什么是反射

每个.class文件被加载到内存后都是一个Class类的对象！例如Demo.class加载到内存后它是Class<Demo>类型的一个对象。



反射就是通过Class对象获取类型相关的信息。一个Class对象就表示一个.class文件，可以通过Class对象获取这个类的构造器、方法，以及成员变量等。

反射是Java的高级特性，在框架中大量被使用！我们必须要了解反射，不然无法学好JavaWeb相关的知识！

1.2　反射相关类

与反射相关的类：

* Class：表示类；
* Field：表示成员变量；
* Method：表示方法；
* Constructor：表示构造器。

###### 2　Class类

2.1　获取Class类

获取Class类的三种基本方式：

* 通过类名称.class，对基本类型也支持；
* Class c = int.class；
* Class c = int[].class；
* Class c = String.class
* 通过对象.getClass()方法
* Class c = obj.getClass();
* Class.forName()通过类名称加载类，这种方法只要有类名称就可以得到Class；
* Class c = Class.forName(“cn.itcast.Demo”);

2.2　Class类的常用方法

* String getName()：获取类名称，包含包名；
* String getSimpleName()：获取类名称，不包含包名；
* Class getSupperClass()：获取父类的Class，例如：new Integer(100).getClass().getSupperClass()返回的是Class<Number>！但new Object().getSupperClass()返回的是null，因为Object没有父类；
* T newInstance()：使用本类无参构造器来创建本类对象；
* boolean isArray()：是否为数组类型；
* boolean isAnnotation()：是否为注解类型；
* boolean isAnnotationPresent(Class annotationClass)：当前类是否被annotationClass注解了；
* boolean isEnum()：是否为枚举类型；
* boolean isInterface()：是否为接口类型；
* boolean isPrimitive()：是否为基本类型；
* boolean isSynthetic()：是否为引用类型；

2.3　通过反射创建对象

|  |
| --- |
| **public** **class** Demo1 {  @Test  **public** **void** fun1() **throws** Exception {  String className = "cn.itcast.User";  Class clazz = Class.*forName*(className);  User user = (User)clazz.newInstance();  System.*out*.println(user);  }  }  **class** User {  **private** String username;  **private** String password;  **public** String getUsername() {  **return** username;  }  **public** **void** setUsername(String username) {  **this**.username = username;  }  **public** String getPassword() {  **return** password;  }  **public** **void** setPassword(String password) {  **this**.password = password;  }  @Override  **public** String toString() {  **return** "User [username=" + username + ", password=" + password + "]";  }  } |
| User [username=null, password=null] |

###### 3　Constructor

Constructor表示一个类的构造器。即构造器的反射对象！

3.1　获取Constructor对象

　　获取Construcator对象需要使用Class对象，下面API来自Class类：

* Constructor getConstructor(Class… parameterTypes)：通过指定的参数类型获取公有构造器反射对象；
* Constructor[] getConstructors()：获取所有公有构造器对象；
* Constructor getDeclaredConstructor(Class… parameterTypes)：通过指定参数类型获取构造器反射对象。可以是私有构造器对象；
* Constructor[] getDeclaredConstructors()：获取所有构造器对象。包含私有构造器；

3.2　Construcator类常用方法

* String getName()：获取构造器名；
* Class getDeclaringClass()：获取构造器所属的类型；
* Class[] getParameterTypes()：获取构造器的所有参数的类型；
* Class[] getExceptionTypes()：获取构造器上声明的所有异常类型；
* T newInstance(Object… initargs)：通过构造器反射对象调用构造器。

3.3　练习：通过Construcator创建对象

User.java

|  |
| --- |
| **public** **class** User {  **private** String username;  **private** String password;  **public** User() {  }  **public** User(String username, String password) {  **this**.username = username;  **this**.password = password;  }  **public** String getUsername() {  **return** username;  }  **public** **void** setUsername(String username) {  **this**.username = username;  }  **public** String getPassword() {  **return** password;  }  **public** **void** setPassword(String password) {  **this**.password = password;  }  @Override  **public** String toString() {  **return** "User [username=" + username + ", password=" + password + "]";  }  } |

Demo1.java

|  |
| --- |
| **public** **class** Demo1 {  @Test  **public** **void** fun1() **throws** Exception {  String className = "cn.itcast.User";  Class clazz = Class.*forName*(className);  Constructor c = clazz.getConstructor(String.**class**, String.**class**);  User user = (User)c.newInstance("zhangSan", "123");  System.*out*.println(user);  }  } |

###### 4　Method(类的组成部分)

Class 🡪 类的反射对象

Constructor 🡪 构造器的反射对象

Method表示方法的反射对象

Field 🡪 成员的反射对象

4.1　获取Method

获取Method需要通过Class对象，下面是Class类的API：

* Method getMethod(String name, Class… parameterTypes)：通过方法名和方法参数类型获取方法反射对象，包含父类中声明的公有方法，但不包含所有私有方法；
* Method[] getMethods()：获取所有公有方法，包含父类中的公有方法，但不包含任何私有方法；
* Method getDeclaredMethod(String name, Class… parameterTypes)：通过方法名和方法参数类型获取本类中声明的方法的反射对象，包含本类中的私有方法，但不包含父类中的任何方法；
* Method[] getDeclaredMethods()：获取本类中所有方法，包含本类中的私有方法，但不包含父类中的任何方法。

4.2　Method常用方法

* String getName()：获取方法名；
* Class getDeclaringClass()：获取方法所属的类型；
* Class[] getParameterTypes()：获取方法的所有参数的类型；
* Class[] getExceptionTypes()：获取方法上声明的所有异常类型；
* Class getReturnType()：获取方法的返回值类型；
* Object invode(Object obj, Object… args)：通过方法反射对象调用方法，如果当前方法是实例方法，那么当前对象就是obj，如果当前方法是static方法，那么可以给obj传递null。args表示是方法的参数；

4.3　练习：通过Method调用方法

|  |
| --- |
| **public** **class** Demo1 {  @Test  **public** **void** fun1() **throws** Exception {  String className = "cn.itcast.User";  Class clazz = Class.*forName*(className);  Constructor c = clazz.getConstructor(String.**class**, String.**class**);  User user = (User)c.newInstance("zhangSan", "123");    Method method = clazz.getMethod("toString");  String result = (String)method.invoke(user);  System.*out*.println(result);  }  } |

###### 5　Field

　　Field表示类的成员变量，可以是实例变量，也可以是静态变量。

5.1　获取Field对象

获取Field对象需要使用Class对象，下面是Class类的API：

* Field getField(String name)：通过名字获取公有成员变量的反射对象，包含父类中声明的公有成员变量；
* Field[] getFields()：获取所有公有成员变量反射对象，包含父类中声明的公有成员变量；
* Field getDeclaredField(String name)：通过名字获取本类中某个成员变量，包含本类的private成员变量，但父类中声明的任何成员变量都不包含；
* Field[] getDeclaredFields()：获取本类中声明的所有成员变量，包含private成员变量，但不包含父类中声明的任何成员变量；

5.2　Field类的常用方法

* String getName()：获取成员变量名；
* Class getDeclaringClass()：获取成员变量的类型；
* Class getType()：获取当前成员变量的类型；
* Object get(Object obj)：获取obj对象的成员变量的值；
* void set(Object obj, Object value)：设置obj对象的成员变量值为value；

5.3　练习：通过Field读写成员

User.java

|  |
| --- |
| **public** **class** User {  **public** String username;  **public** String password;  **public** User() {  }  **public** User(String username, String password) {  **this**.username = username;  **this**.password = password;  }  **public** String getUsername() {  **return** username;  }  **public** **void** setUsername(String username) {  **this**.username = username;  }  **public** String getPassword() {  **return** password;  }  **public** **void** setPassword(String password) {  **this**.password = password;  }  @Override  **public** String toString() {  **return** "User [username=" + username + ", password=" + password + "]";  }  } |

Demo1.java

|  |
| --- |
| **public** **class** Demo1 {  @Test  **public** **void** fun1() **throws** Exception {  String className = "cn.itcast.User";  Class clazz = Class.*forName*(className);  User user = **new** User("zhangSan", "123");    Field field1 = clazz.getField("username");  Field field2 = clazz.getField("password");    String username = (String)field1.get(user);  String password = (String)field2.get(user);    System.*out*.println(username + ", " + password);    field1.set(user, "liSi");  field2.set(user, "456");    System.*out*.println(user);  }  } |

###### 6　 AccessibleObject

AccessibleObject类是Constructor、Method、Field三个类的父类。AccessibleObject最为重要的方法如下：

* boolean isAccessible()：判断当前成员是否可访问；
* void setAccessible(boolean flag)：设置当前成员是否可访问。

当Constructor、Method、Field为私有时，如果我们想反射操作，那么就必须先调用反射对象的setAccessible(true)方法，然后才能操作。

User.java

|  |
| --- |
| **public class** User {  **private** String username;  **private** String password;  **public** User() {  }  **public** User(String username, String password) {  **this**.username = username;  **this**.password = password;  }  **public** String getUsername() {  **return** username;  }  **public** **void** setUsername(String username) {  **this**.username = username;  }  **public** String getPassword() {  **return** password;  }  **public** **void** setPassword(String password) {  **this**.password = password;  }  @Override  **public** String toString() {  **return** "User [username=" + username + ", password=" + password + "]";  }  } |

注意，User类的username和password成员变量为private的，这时再通过Field来反射操作这两个成员变量就必须先通过setAccessible(true)设置后才行。

|  |
| --- |
| **public** **class** Demo1 {  @Test  **public** **void** fun1() **throws** Exception {  String className = "cn.itcast.User";  Class clazz = Class.*forName*(className);  User user = **new** User("zhangSan", "123");    Field field1 = clazz.getDeclaredField("username");  Field field2 = clazz.getDeclaredField("password");    field1.setAccessible(**true**);  field2.setAccessible(**true**);    String username = (String)field1.get(user);  String password = (String)field2.get(user);    System.*out*.println(username + ", " + password);    field1.set(user, "liSi");  field2.set(user, "456");    System.*out*.println(user);  }  } |