10: 检测类型

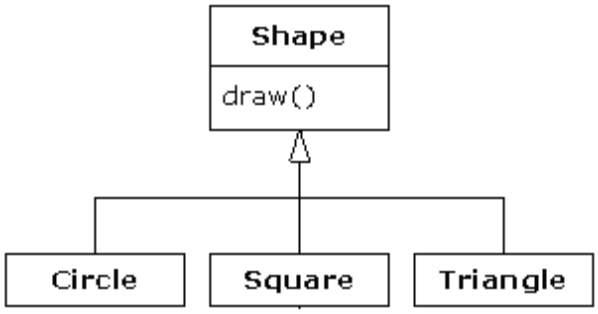
初看起来“运行时类型识别(run-time type identification，缩写为RTTI)”的想法很简单：要让你在只持有这个对象的基类的 reference 的情况下，找出它的确切的类型。

但是，这种对“RTTI 的强烈需求”揭示了许多 OO 设计中会碰到的有趣(同时也很令人困惑)的问题，并且引出了“该如何组织程序结构”这一根本性的问题。

本章要讲解 Java 所提供的，让你在运行时发现对象和类的信息的方法。它有两种形式：一种是假设你在编译和运行时都完全知道类型的具体信息的“传统的”RTTI，另一种是允许你只依靠运行时信息发掘类型信息的“reflection”机制。我们先讲“传统的”RTTI，再讨论 reflection。

为什么会需要 RTTI

先想一想我们现在已经很熟悉的那个多态性的例子。通用的基类 **Shape** 派生出具体的 **Circle**，**Square** 和 **Triangle** 类：



这是一个典型的类系(class hierarchy)结构图，基类在最上面，派生类在下面。OOP 的目的就是要让你用基类的 reference(这里就是 **Shape**)来写程序，因此如果你往程序里面加了一个新的类(比如 **Shape** 又派生了一个 **Rhomboid**)，绝大多数的代码根本不会受到影响。在这个例子里面，**Shape** 接口动态绑定的是 **draw( )**，于是客户程序员能够通过通用的 **Shape** reference 来调用 **draw( )**。所有派生类都会覆写 **draw( )**，而且由于这个方法是动态绑定的，因此即便是通过通用的 **Shape** reference 来进行的调用，也能得到正确的行为。这就是多态性。

由此，编程时一般会先创建一个具体的对象(**Circle**，**Square**，或 **Triangle**)，再把它上传到 **Shape**(这样就把对象的具体类型给忘了)，然后在余下来的程序里面使用那个匿名的 **Shape** reference。

www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

作为多态性的简要回顾，我们再用程序来描述一遍上面这个例子：

```
//: c10:Shapes.java
import com.bruceeckel.simpletest.*;

class Shape {
  void draw() { System.out.println(this +
".draw()"); }
}

class Circle extends Shape {
  public String toString() { return "Circle"; }
}

class Square extends Shape {
  public String toString() { return "Square"; }
}

class Triangle extends Shape {
  public String toString() { return "Triangle"; }
}

public class Shapes {
  private static Test monitor = new Test();
  public static void main(String[] args) {
    // Array of Object, not Shape:
    Object[] shapeList = {
      new Circle(),
      new Square(),
      new Triangle()
    };
    for(int i = 0; i < shapeList.length; i++)
      ((Shape)shapeList[i]).draw(); // Must cast
    monitor.expect(new String[] {
      "Circle.draw()",
      "Square.draw()",
      "Triangle.draw()"
    });
  }
} ///:~
```

基类里有一个间接调用 **toString( )**来打印类的打印标识符的 **draw( )** 方法。它把 **this** 传给 **System.out.println( )**，而这个方法一看到对象就会自动调用它的 **toString( )**方法，这样就能生成对象的 **String** 表达形式了。每个派生类都会覆写 **toString( )**方法(从 **Object** 继承下来的)，这样 **draw( )**就能(多态地)打印各种对象了。

main( )创建了一些具体的 **Shape** 对象并且把它们加进一个数组。这个数组有点怪，因为它不是 **Shape**(尽管可以这样)，而是根类 **Object** 的数组。这是在为第 11 章要讲的 *collection*(也称 *container*)作铺垫。collection 是一种工具，它只有一种用途，就是要为你保管其它对象。因此出于通用性的考虑，这些 collection 应该能持有任何东西。所以它们持

www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

有 **Object**。因此你可以从 **Object** 数组看到一个你将在第 11 章遇到的重要问题。

在这个例子里，当把 **Shape** 放进 **Object** 数组的时候，上传就发生了。由于 Java 里面的所有东西(除了 primitive)都是对象，因此 **Object** 数组也能持有 **Shape** 对象。但是在上传过程中，“这个对象是 **Shape**”的信息丢失了。对数组而言，它们都只是 **Object**。

但是，当你用下标把元素从数组里提取出来的时候，就免不了要忙活一阵了。由于数组持有的都是些 **Object**，因此提取出来的也就自然都是 **Object** 的 reference 了。但是我们知道，它实际上是 **Shape** 的 reference，而且我们也要向它发送 **Shape** 的消息。于是老套的“**(Shape)**”类型转换就变得必不可少了。这是 RTTI 的最基本的形式，因为程序运行的时候会逐项检查转换是不是都正确。这种检查正体现了 RTTI 的本义：在运行时鉴别对象的类型。

这里的 RTTI 转换并不彻底：**Object** 只是转换成 **Shape**，而不是一下子转换成 **Circle**，**Square** 或 **Triangle**。这是因为现阶段我们**只知道**数组存储的是 **Shape**。编译的时候，这种转换要由你自己来把关，但是运行的时候这种转换就不用你操心了。

接下来就交给多态性了。**Shape** 到底要执行哪段代码，这要由 reference 究竟是指向 **Circle**，**Square** 还是 **Triangle** 对象来决定。总之程序就该这么写；你的目标就是，要让绝大多数代码只同代表这个类系的基类对象打交道(在本案中，就是 **Shape**)，对它们来说，越少知道下面的具体类型越好。这样一来，代码的读写和维护就会变得更简单，而实现，理解和修改设计方案也会变得更容易一些。所以多态性是 OOP 的一个主要目标。

但是如果你碰到一个特殊问题，而要解决这个问题，最简单的办法就是“找出这个通用的 reference 究竟是指哪个具体类型的”，那你又该怎么办呢？比方说，你要让用户能用“把这类对象全都染成紫色”的办法来突出某种 **Shape**。或者，你要写一个会用到“旋转”一组 shape 的方法。但是对圆形来说，旋转是没有意义的，所以你只想跳过圆形对象。有了 RTTI，你就能问出 **Shape** 的 reference 究竟是指的哪种具体类型的对象了，这样你就能把特例给区别出来了。

Class 对象

想要知道 Java 的 RTTI 是如何工作的，你就必须首先知道程序运行的时候，类型信息是怎样表示的。这是由一种特殊的，保存类的信息的，叫作“*Class* 对象 (*Class object*)”的对象来完成的。实际上类的“常规”对象是由 **Class** 对象创建的。

www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

程序里的每个类都要有一个 **Class** 对象。也就是说，每次你撰写并且编译了一个新的类的时候，你就创建了一个新的 **Class** 对象(而且可以这么说，这个对象会存储在同名的**.class** 文件里)。程序运行时，当你需要创建一个那种类的对象的时候，JVM 会检查它是否装载了那个 **Class** 对象。如果没有，JVM 就会去找那个**.class** 文件，然后装载。由此也可知道，Java 程序在启动的时候并没有完全装载，这点同许多传统语言是不一样的。

一旦那种类型的 **Class** 对象被装进了内存，所有那个类的对象就都会由它来创建了。如果这听上去太玄，或者你不相信的话，下面这个程序就能作证明：

```
//: c10:SweetShop.java
// Examination of the way the class loader works.
import com.bruceeckel.simpletest.*;

class Candy {
  static {
    System.out.println("Loading Candy");
  }
}

class Gum {
  static {
    System.out.println("Loading Gum");
  }
}

class Cookie {
  static {
    System.out.println("Loading Cookie");
  }
}

public class SweetShop {
  private static Test monitor = new Test();
  public static void main(String[] args) {
    System.out.println("inside main");
    new Candy();
    System.out.println("After creating Candy");
    try {
      Class.forName("Gum");
    } catch(ClassNotFoundException e) {
      System.out.println("Couldn't find Gum");
    }
    System.out.println("After
Class.forName(\"Gum\")");
    new Cookie();
    System.out.println("After creating Cookie");
    monitor.expect(new String[] {
      "inside main",
      "Loading Candy",
      "After creating Candy",
      "Loading Gum",
      "After Class.forName(\"Gum\")",
      "Loading Cookie",
```


www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

```
        "After creating Cookie"
      });
    }
} ///:~
```

Candy，**Gum**，和 **Cookie**，每个类都有一条“会在类第一次被装载的时候执行”的 **static** 语句。这样装载的类时候就会它就会打印消息通知你了。对象的创建被分散到了 **main( )**的各条打印语句之间，其目的就是要帮助你检测装载的时机。

你可以从程序的输出看到，**Class** 对象只会在需要的时候装载，而 **static** 初始化则发生在装载类的时候。

下面这行特别有趣：

```
Class.forName("Gum");
```

这是一个 **Class** 的 **static** 方法(所有的 **Class** 对象所共有的)。**Class** 对象同其它对象一样，也可以用 reference 来操控(这是装载器要干的)，而要想获取其 reference，**forName( )**就是一个办法。它要一个表示这个类的名字的 **String** 作参数(一定要注意拼写和大小写！)。 这个方法会返回 **Class** 的 reference，不过程序里面没用到这个 reference；这里只是要用它的副作用——看看 **Gum** 类装载了没有，要是还没有那就马上装载。装载的过程中，程序执行了 **Gum** 的 **static** 语句。

在上述例程中，如果 **Class.forName( )**没有找到它要装载的类，就会抛出一个 **ClassNotFoundException**(太完美了！单从异常的名字你就能知道出了什么问题了)。这里，我们只是简单地报告一下问题，然后继续下去，但是在较为复杂的程序里，你或许应该在异常处理程序里把这个问题解决掉。

Class 常数

Java 还提供了一种获取 **Class** 对象的 reference 的方法：“*class* 常数 (*class literal*)”。对于上述程序，它就是：

```
Gum.class;
```

这种写法不但更简单，而且也更安全，因为它是在编译时做检查的。此外由于没有方法调用，它的执行效率也更高一些。

www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

Class 常数不但能用于普通类，也可以用于接口，数组和 primitive 类型。此外，每种 primitive 的 wrapper 类还有一个标准的，名为 **TYPE** 的数据成员。这个 **TYPE** 能返回“与这种 primitive 相关联的 wrapper 类”的 **Class** 对象的 reference，就像这样：

... 等同于...	
boolean.class	**Boolean.TYPE**
char.class	**Character.TYPE**
byte.class	**Byte.TYPE**
short.class	**Short.TYPE**
int.class	**Integer.TYPE**
long.class	**Long.TYPE**
float.class	**Float.TYPE**
double.class	**Double.TYPE**
void.class	**Void.TYPE**

我喜欢尽量使用“**.class**”，因为这种写法能与普通类的保持一致。

转换之前先作检查

到目前为止，你看到的 RTTI 的形式有：

1. 经典的类型转换；如**“(Shape)”**，这种转换要经过 RTTI 的检查。要是做了错误的转换，它就会抛出 **ClassCastException**。
2. 代表对象类型的 **Class** 对象。你可以在运行的时候查询 Class 对象，以此来获取所需的信息。

在 C++里，经典的“**(Shape)**”形式的转换并不动用 RTTI。它只是简单地告诉编译器把这个对象当作那个新的类型来用。但 Java 会作类型检查，所以 Java 的类型转换常常被称作“类型安全的下传”。之所以要说“下传”，是因为在继承关系图里，派生类一般会放在下面。如果把 **Circle** 到 **Shape** 的转换称作上传，那么 **Shape** 到 **Circle** 的转换就应该叫下传了。不过你知道 **Circle** 就是 **Shape**，而且编译器也不会阻止去上传，但是你不一定知道 **Shape** 是不是一个 **Circle**，所以如果不进行明确的类型转换的话，编译器是不会让你把对象赋给派生类的 reference 的。

Java 里面还有第三种 RTTI 的形式。这就是 **instanceof** 关键词，它会告诉你对象是不是某个类的实例。它返回的是一个 **boolean** 值，因此你就可以用提问的形式来用了，就像这样：

```
if(x instanceof Dog)
```


www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

```
((Dog)x).bark();
```

在将 **x** 转换成 **Dog** 之前，**if** 语句会先看看 **x** 对象是不是 **Dog** 类的。如果没有其它信息能告诉你这个对象的类型，那么在下传之前先用 **instanceof** 检查一下是很重要的；否则的话，你就很有可能会撞上 **ClassCastException**。

通常情况下，你只是要找一种类型(比如把三角形都变成紫色的)，但是你也可以用 **instanceof** 标出所有对象的类型。假设你有一个 **Pet** 类系：

```
//: c10:Pet.java
package c10;
public class Pet {} ///:~
```

```
//: c10:Dog.java
package c10;
public class Dog extends Pet {} ///:~
```

```
//: c10:Pug.java
package c10;
public class Pug extends Dog {} ///:~
```

```
//: c10:Cat.java
package c10;
public class Cat extends Pet {} ///:~
```

```
//: c10:Rodent.java
package c10;
public class Rodent extends Pet {} ///:~
```

```
//: c10:Gerbil.java
package c10;
public class Gerbil extends Rodent {} ///:~
```

```
//: c10:Hamster.java
package c10;
public class Hamster extends Rodent {} ///:~
```


www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

下面，我们要找出各种 **Pet** 的数量，所以我们要用一个带 **int** 的类来保存这个信息。你可以把它当成是一个可以修改的 **Integer**：

```
//: c10:Counter.java
package c10;

public class Counter {
  int i;
  public String toString() { return
Integer.toString(i); }
} ///:~
```

下一步，我们需要一个能同时持有两种对象的工具：一个表示 **Pet** 的类型，另一个是表示宠物数量的 **Counter**。也就是说，我们要问“这里有多少 **Gerbil** 对象？”普通数组作不了这个，因为它是用下标来定位的。我们要的是能用 **Pet** 的类型来定位的数组。我们要将 **Counter** 对象同 **Pet** 对象关联起来。为了能准确的做到这一点，我们会用到一种被称为“关联性数组(*associative array*)”的标准数据结构。下面是它最简单的实现：

```
//: c10:AssociativeArray.java
// Associates keys with values.
package c10;
import com.bruceeckel.simpletest.*;

public class AssociativeArray {
  private static Test monitor = new Test();
  private Object[][] pairs;
  private int index;
  public AssociativeArray(int length) {
    pairs = new Object[length][2];
  }
  public void put(Object key, Object value) {
    if(index >= pairs.length)
      throw new ArrayIndexOutOfBoundsException();
    pairs[index++] = new Object[] { key, value };
  }
  public Object get(Object key) {
    for(int i = 0; i < index; i++)
      if(key.equals(pairs[i][0]))
        return pairs[i][1];
    throw new RuntimeException("Failed to find key");
  }
  public String toString() {
    String result = "";
    for(int i = 0; i < index; i++) {
      result += pairs[i][0] + " : " + pairs[i][1];
      if(i < index - 1) result += "\n";
    }
    return result;
  }
```


www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

```
  public static void main(String[] args) {
    AssociativeArray map = new AssociativeArray(6);
    map.put("sky", "blue");
    map.put("grass", "green");
    map.put("ocean", "dancing");
    map.put("tree", "tall");
    map.put("earth", "brown");
    map.put("sun", "warm");
    try {
      map.put("extra", "object"); // Past the end
    } catch(ArrayIndexOutOfBoundsException e) {
      System.out.println("Too many objects!");
    }
    System.out.println(map);
    System.out.println(map.get("ocean"));
    monitor.expect(new String[] {
      "Too many objects!",
      "sky : blue",
      "grass : green",
      "ocean : dancing",
      "tree : tall",
      "earth : brown",
      "sun : warm",
      "dancing"
    });
  }
} ///:~
```

如果是第一次看到这个程序，你会觉得这好像是一个通用工具，应该把它放进 **com.bruceeckel.tools**。是的，这的确是一个通用工具——太有用了，以至于 **java.util** 提供了好几种关联性数组(其正式的命名是 Map)，它们的功能比这可强多了，速度也快了许多。第 11 章会用大量的篇幅讲解关联性数组。但是它们太复杂了，所以这里临时作了一个，目的就是要让你在理解关联性数组的价值的同时，不至于让问题变得太复杂。

在关联性数组中，下标被称为键(*key*)，而与之相关联的对象则被称为值(*value*) 。这里，我们用“构建一个双元素数组的 **pair** 数组，并且用它来保存键和值”的方法，建立了键值之间的关联。构造函数创建的是一个定长数组，因此你得用 **index** 来确保它不会过界。**put( )**一对新的键值时，它会创建一个新的双元素数组，并且把它放到 **pairs** 的下一个空位里。如果 **index** 大于等于 **pairs** 的长度，它就抛出异常。

要用 **get( )**的时候，你只要把 **key** 当参数传给它就可以了，它会帮你找出值，然后把结果返回给你，要是找不到，它就抛异常。**get( )**用了一种你能想到的最慢的算法：从数组的开头，用 **equals( )**作比较。但是这里更看重简单而不是效率，而且第 11 章要将的 Map 已经解决了这个问题，因此我们不必担心。

www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

put( )和 **get( )**是关联性数组的两个基本方法，但是考虑到程序的输出，我们还覆写了 **toString( )**方法，让它把键和值全都打印出来。为了看看它是不是能正常工作，**main( )**先是往 **AssociativeArray** 里填了一组成对的字符串，然后把它打印了出来，最有再用 **get( )**提取了一个值。

现在所有工具都已经准备好了，我们可以用 **instanceof** 来数 **Pet** 了：

```
//: c10:PetCount.java
// Using instanceof.
package c10;
import com.bruceeckel.simpletest.*;
import java.util.*;

public class PetCount {
  private static Test monitor = new Test();
  private static Random rand = new Random();
  static String[] typenames = {
    "Pet", "Dog", "Pug", "Cat",
    "Rodent", "Gerbil", "Hamster",
  };
  // Exceptions thrown to console:
  public static void main(String[] args) {
    Object[] pets = new Object[15];
    try {
      Class[] petTypes = {
        Class.forName("c10.Dog"),
        Class.forName("c10.Pug"),
        Class.forName("c10.Cat"),
        Class.forName("c10.Rodent"),
        Class.forName("c10.Gerbil"),
        Class.forName("c10.Hamster"),
      };
      for(int i = 0; i < pets.length; i++)
        pets[i] =
petTypes[rand.nextInt(petTypes.length)]
          .newInstance();
    } catch(InstantiationException e) {
      System.out.println("Cannot instantiate");
      System.exit(1);
    } catch(IllegalAccessException e) {
      System.out.println("Cannot access");
      System.exit(1);
    } catch(ClassNotFoundException e) {
      System.out.println("Cannot find class");
      System.exit(1);
    }
    AssociativeArray map =
      new AssociativeArray(typenames.length);
    for(int i = 0; i < typenames.length; i++)
      map.put(typenames[i], new Counter());
    for(int i = 0; i < pets.length; i++) {
      Object o = pets[i];
      if(o instanceof Pet)
        ((Counter)map.get("Pet")).i++;
      if(o instanceof Dog)
        ((Counter)map.get("Dog")).i++;
      if(o instanceof Pug)
```


www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

```
            ((Counter)map.get("Pug")).i++;
          if(o instanceof Cat)
            ((Counter)map.get("Cat")).i++;
          if(o instanceof Rodent)
            ((Counter)map.get("Rodent")).i++;
          if(o instanceof Gerbil)
            ((Counter)map.get("Gerbil")).i++;
          if(o instanceof Hamster)
            ((Counter)map.get("Hamster")).i++;
        }
        // List each individual pet:
        for(int i = 0; i < pets.length; i++)
          System.out.println(pets[i].getClass());
        // Show the counts:
        System.out.println(map);
        monitor.expect(new Object[] {
          new TestExpression("%% class c10\\."+
            "(Dog|Pug|Cat|Rodent|Gerbil|Hamster)",
            pets.length),
          new TestExpression(
            "%% (Pet|Dog|Pug|Cat|Rodent|Gerbil|Hamster)"
+
            " : \\d+", typenames.length)
        });
      }
    } ///:~
```

main( )用 **Class.forName( )**创建了一个名为 **petTypes** 的 **Class** 对象的数组。由于 **Pet** 对象属于 **c10** 这个 package，因此命名的时候要把 package 的名字也包括进去。

接下来就要把 **pets** 数组填满了。这里用的方法是，先在 **petTypes** 数组里面随机选取一个 **Class** 对象，再调用它的 **Class.newInstance( )** 方法，而这个方法会调用类的默认(也就是无参数的)构造函数，这样就能生成一个新的对象了。

forName( )和 **newInstance( )**都会产生异常，或许你已经从 **try** 及其后面的 **catch** 语句看出了这一点。此外，异常名再一次向我们解释了出了什么错。(**IllegalAccessException** 与违反 Java 的安全机制有关)。

创建完 **AssociativeArray** 之后，我们再用 **Pet** 和 **Counter** 的键值组合把它填满。接下来就要用 **instanceof** 来测试数组中的各个元素，并且为它们计数了。最后，我们还要打印 **pets** 数组和 **AssociativeArray**，这样你就能比较结果了。

instanceof 的限制很严：你只能拿它同类的名字，而不是 **Class** 对象作比较。可能你觉得要写那么多 **instanceof** 实在是太无聊了，确实如此。但是“先创建一个 **Class** 对象的数组，再拿它来作比较”的方法，看上去是挺聪明的，实际上却行不通(别走开，你马上就能看到替代方案

www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

了)。但是这个限制也没有你想的那么严重，因为当你发现 **instanceof** 太多的时候，你就知道设计方案有问题了。

当然这个例子有些做作——你完全可以在类里放一个 **static** 成员，然后用构造函数去递增它，这样就能掌控对象的数目了。要是你能看到这个类的源代码的话，你就可以用这个方法来作修改了。不过情况不总是这样，所以我们需要 RTTI。

使用类常数

能看看怎样用类常数来重写 **PetCount.java** 也是很有意思的。而且重写之后，程序看上去更清楚了。

```
//: c10:PetCount2.java
// Using class literals.
package c10;
import com.bruceeckel.simpletest.*;
import java.util.*;

public class PetCount2 {
  private static Test monitor = new Test();
  private static Random rand = new Random();
  public static void main(String[] args) {
    Object[] pets = new Object[15];
    Class[] petTypes = {
      // Class literals:
      Pet.class,
      Dog.class,
      Pug.class,
      Cat.class,
      Rodent.class,
      Gerbil.class,
      Hamster.class,
    };
    try {
      for(int i = 0; i < pets.length; i++) {
        // Offset by one to eliminate Pet.class:
        int rnd = 1 + rand.nextInt(petTypes.length -
1);
        pets[i] = petTypes[rnd].newInstance();
      }
    } catch(InstantiationException e) {
      System.out.println("Cannot instantiate");
      System.exit(1);
    } catch(IllegalAccessException e) {
      System.out.println("Cannot access");
      System.exit(1);
    }
    AssociativeArray map =
      new AssociativeArray(petTypes.length);
    for(int i = 0; i < petTypes.length; i++)
      map.put(petTypes[i].toString(), new Counter());
    for(int i = 0; i < pets.length; i++) {
      Object o = pets[i];
      if(o instanceof Pet)
        ((Counter)map.get("class c10.Pet")).i++;
```


www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

```
        if(o instanceof Dog)
          ((Counter)map.get("class c10.Dog")).i++;
        if(o instanceof Pug)
          ((Counter)map.get("class c10.Pug")).i++;
        if(o instanceof Cat)
          ((Counter)map.get("class c10.Cat")).i++;
        if(o instanceof Rodent)
          ((Counter)map.get("class c10.Rodent")).i++;
        if(o instanceof Gerbil)
          ((Counter)map.get("class c10.Gerbil")).i++;
        if(o instanceof Hamster)
          ((Counter)map.get("class c10.Hamster")).i++;
      }
      // List each individual pet:
      for(int i = 0; i < pets.length; i++)
        System.out.println(pets[i].getClass());
      // Show the counts:
      System.out.println(map);
      monitor.expect(new Object[] {
        new TestExpression("%% class c10\\." +
          "(Dog|Pug|Cat|Rodent|Gerbil|Hamster)",
          pets.length),
        new TestExpression("%% class c10\\." +
          "(Pet|Dog|Pug|Cat|Rodent|Gerbil|Hamster) :
\\d+",
          petTypes.length)
      });
    }
} ///:~
```

这里我们想试试直接从 **Class** 对象那里得到类的名字，因此把 **typenames** 数组去掉了。注意，系统可以分辨哪些是类，哪些是接口。

此外，创建 **petTypes** 的时候已经不需要用 **try** 了。编译的时候已经做过检查了，因此它不会像 **Class.forName( )**那样抛出异常的。

再有，动态创建 **Pet** 对象的时候，生成的随机数是有限制的，它应该小于 **petTypes.length** 而且不包括零。这是因为零代表了 **Pet.class**，而我们对泛型的 **Pet** 对象不感兴趣。但是，由于 **petTypes** 也包括 **Pet.class**，所以计数的时候也把 **Pet** 算了一遍。

动态的 instanceof

Class.isInstance 方法还提供了一种动态调用 **instanceof** 的方法。这样，**PetCount** 里面那些丑陋的 **instanceof** 就都能被去除了。

```
//: c10:PetCount3.java
// Using isInstance()
package c10;
import com.bruceeckel.simpletest.*;
import java.util.*;
```


www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

```
public class PetCount3 {
  private static Test monitor = new Test();
  private static Random rand = new Random();
  public static void main(String[] args) {
    Object[] pets = new Object[15];
    Class[] petTypes = {
      // Class literals:
      Pet.class,
      Dog.class,
      Pug.class,
      Cat.class,
      Rodent.class,
      Gerbil.class,
      Hamster.class,
    };
    try {
      for(int i = 0; i < pets.length; i++) {
        // Offset by one to eliminate Pet.class:
        int rnd = 1 + rand.nextInt(petTypes.length -
1);
        pets[i] = petTypes[rnd].newInstance();
      }
    } catch(InstantiationException e) {
      System.out.println("Cannot instantiate");
      System.exit(1);
    } catch(IllegalAccessException e) {
      System.out.println("Cannot access");
      System.exit(1);
    }
    AssociativeArray map =
      new AssociativeArray(petTypes.length);
    for(int i = 0; i < petTypes.length; i++)
      map.put(petTypes[i].toString(), new Counter());
    for(int i = 0; i < pets.length; i++) {
      Object o = pets[i];
      // Using Class.isInstance() to eliminate
      // individual instanceof expressions:
      for(int j = 0; j < petTypes.length; ++j)
        if(petTypes[j].isInstance(o))

((Counter)map.get(petTypes[j].toString())).i++;
    }
    // List each individual pet:
    for(int i = 0; i < pets.length; i++)
      System.out.println(pets[i].getClass());
    // Show the counts:
    System.out.println(map);
    monitor.expect(new Object[] {
      new TestExpression("%% class c10\\." +
        "(Dog|Pug|Cat|Rodent|Gerbil|Hamster)",
        pets.length),
      new TestExpression("%% class c10\\." +
        "(Pet|Dog|Pug|Cat|Rodent|Gerbil|Hamster) :
\\d+",
        petTypes.length)
    });
  }
} ///:~
```


www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

可以这么说，**isInstance( )**能完全替代 **instanceof**。此外，它还能让你用修改 **petTypes** 数组的方式添加新的 **Pet** 类型；程序的其余部分无需改动(而用 **instanceof** 的话，就不得不作修改了)。

instanceof vs. Class 的相等性

有两种方法可以查出对象的类型信息，一是 **instanceof**(包括 **instanceof** 和 **isInstance( )**，它们效果是相同的)，另一个就是比较 **Class** 对象，这两者之间有着重大的区别。下面这段程序就向你演示了它们之间的区别：

```
//: c10:FamilyVsExactType.java
// The difference between instanceof and class
package c10;
import com.bruceeckel.simpletest.*;

class Base {}
class Derived extends Base {}

public class FamilyVsExactType {
  private static Test monitor = new Test();
  static void test(Object x) {
    System.out.println("Testing x of type " +
      x.getClass());
    System.out.println("x instanceof Base " +
      (x instanceof Base));
    System.out.println("x instanceof Derived " +
      (x instanceof Derived));
    System.out.println("Base.isInstance(x) " +
      Base.class.isInstance(x));
    System.out.println("Derived.isInstance(x) " +
      Derived.class.isInstance(x));
    System.out.println("x.getClass() == Base.class "
+
      (x.getClass() == Base.class));
    System.out.println("x.getClass() ==
Derived.class " +
      (x.getClass() == Derived.class));

System.out.println("x.getClass().equals(Base.class))
"+
      (x.getClass().equals(Base.class)));
    System.out.println(
      "x.getClass().equals(Derived.class)) " +
      (x.getClass().equals(Derived.class)));
  }
  public static void main(String[] args) {
    test(new Base());
    test(new Derived());
    monitor.expect(new String[] {
      "Testing x of type class c10.Base",
      "x instanceof Base true",
      "x instanceof Derived false",
      "Base.isInstance(x) true",
```


www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

```
        "Derived.isInstance(x) false",
        "x.getClass() == Base.class true",
        "x.getClass() == Derived.class false",
        "x.getClass().equals(Base.class)) true",
        "x.getClass().equals(Derived.class)) false",
        "Testing x of type class c10.Derived",
        "x instanceof Base true",
        "x instanceof Derived true",
        "Base.isInstance(x) true",
        "Derived.isInstance(x) true",
        "x.getClass() == Base.class false",
        "x.getClass() == Derived.class true",
        "x.getClass().equals(Base.class)) false",
        "x.getClass().equals(Derived.class)) true"
      });
  }
} ///:~
```

test( )方法用了两种形式的 **instanceof** 来检查参数的类型。接着它获取了 **Class** 对象的 reference，然后用==和 **equals( )**测试其相等性。值得欣慰的是，**instanceof** 同 **isInstance( )**得出的结果是相同的，**equals( )**和==也一样。但是这两组这两组测试所得出的结论却是不同的。**instanceof** 遵循了类型的概念，它问的是“你是这个类，或者是从这个类派生出的类吗？”而用 == 去比较两个 **Class** 对象就与继承无关了——它要么是这个具体类型的，要么就不是。

RTTI 的语法

Java 要用 **Class** 对象来进行 RTTI，甚至类型转换也要用到它。此外，**Class** 类还提供了很多其它与 RTTI 相关的用途。

首先，你必须得到所需的 **Class** 对象的 reference。一种办法是在前面例程中演示的将一个字符串传给 **Class.forName( )**方法。这个方法很方便，因为这样一来，你就不再需要为获取某个类的 **Class** 对象的 reference 而去事先准备一个那个类的对象了。但是如果你已经有一个那种类的对象的话，你就能用 **getClass( )**来获取 **Class** 的 reference 了。这是 **Object** 根类的方法，它会返回对象所属的真实类型的 **Class** 对象的 reference。下面这段程序演示了很多有趣的 **Class** 类的方法。

```
//: c10:ToyTest.java
// Testing class Class.
import com.bruceeckel.simpletest.*;

interface HasBatteries {}
interface Waterproof {}
interface Shoots {}
class Toy {
  // Comment out the following default constructor
  // to see NoSuchMethodError from (*1*)
  Toy() {}
```


www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

```
  Toy(int i) {}
}

class FancyToy extends Toy
implements HasBatteries, Waterproof, Shoots {
  FancyToy() { super(1); }
}

public class ToyTest {
  private static Test monitor = new Test();
  static void printInfo(Class cc) {
    System.out.println("Class name: " + cc.getName()
+
      " is interface? [" + cc.isInterface() + "]");
  }
  public static void main(String[] args) {
    Class c = null;
    try {
      c = Class.forName("FancyToy");
    } catch(ClassNotFoundException e) {
      System.out.println("Can't find FancyToy");
      System.exit(1);
    }
    printInfo(c);
    Class[] faces = c.getInterfaces();
    for(int i = 0; i < faces.length; i++)
      printInfo(faces[i]);
    Class cy = c.getSuperclass();
    Object o = null;
    try {
      // Requires default constructor:
      o = cy.newInstance(); // (*1*)
    } catch(InstantiationException e) {
      System.out.println("Cannot instantiate");
      System.exit(1);
    } catch(IllegalAccessException e) {
      System.out.println("Cannot access");
      System.exit(1);
    }
    printInfo(o.getClass());
    monitor.expect(new String[] {
      "Class name: FancyToy is interface? [false]",
      "Class name: HasBatteries is interface?
[true]",
      "Class name: Waterproof is interface? [true]",
      "Class name: Shoots is interface? [true]",
      "Class name: Toy is interface? [false]"
    });
  }
} ///:~
```

FancyToy 类相当复杂，它继承了 **Toy**，实现了 **HasBatteries**，**Waterproof** 和 **Shoots** 接口。**main( )**里边先创建一个 **Class** 的 reference，然后用“由 **try** 括起来的” **forName( )**方法把它初始化为 **FancyToy** 的 **Class** 对象。

www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

Class.getInterfaces( )方法会返回一个 **Class** 对象的数组。数组中的对象分别表示了它所实现的接口。

如果你手上有一个 **Class** 对象，你还能用 **getSuperclass( )**问出它最近的那个父类。当然，这会返回一个 **Class** 的 reference，于是你可以接着问。也就是说，程序运行的时候，你能以此发现对象的完整的类系。

初看起来，**Class** 的 **newInstance( )**方法就像是另一种 **clone( )**对象的方法。但是，你却可以用 **newInstance( )**凭空创建出一个新的对象。就像这里所看到的，程序里面没有 **Toy** 对象——只有一个指向“**Toy** 类的 **Class** 对象”的 reference，**cy**。这是一种实现“虚拟构造函数(virtual constructor)”的方法，它能让你写出“我不管你到底是什么类的，但是你一定要把你自己创建好”的代码。在上述程序中， **cy** 只是一个 **Class** 对象的 reference，因此编译器得到的信息非常有限。所以当方法返回对象的时候，你只能把它交给 **Object** 的 reference。但是这个 reference 指的却是一个 **Toy** 的对象。当然，在你能向它发送“除 **Object** 接口以外”的消息之前，还得调查一番，并做一个类型转换。此外，用 **newInstance( )**创建的对象的前提是，这个类还必须要有默认构造函数。下一节，我们会讲 Java 的 *reflection* API，届时你就能用任意一个构造函数来动态的创建对象了。

这份清单的最后一项就是 **printInfo( )**方法。它拿一个 **Class** 对象的 reference 作参数，用 **getName( )**提取类的名字，用 **isInterface( )** 判断它是不是接口。这样，你就能仅凭 **Class** 对象就找出所有你想知道的这个对象的信息了。

Reflection: 运行时的类信息

如果你不知道对象的确切类型，RTTI 会告诉你。但是它有一个前提：为了让 RTTI 能找出这些类型，并且利用这些信息，这些类必须是编译时已知的。换言之，要想让 RTTI 正常运行，编译器必须知道所有的类。

初看起来，这也不算是什么限制，但是如果你得到一个不在程序管辖范围之内的对象的 reference，那又该怎么办呢。实际上，程序编译的时候，根本得不到这个对象的 class 文件。比方说，你拿到一张磁盘，或者打开了一个网络连接，然后你被告知这些数据代表某个类。但是编译程序的时候，编译器不可能知道将来出现的类，那么你该怎样使用这个类呢？

在传统的编程环境中，这种假设有点牵强。但是当我们转向更大的编程世界的时候，就会发现这种事情会出现在一些很重要的地方。首先要碰到的就是“基于组件的编程(component-based programming)”，也就是用 *Rapid Application Development*(RAD)工具搞开发时所用的技术。这是一种，“通过将代表组件的图标拖到窗体来创建程序的”可视化编程方式(在这种开发方式下，你在屏幕上看到的就是“窗体”)。然后，编程

www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

的时候就要设置组件的值，并以此来进行配置。设计阶段的配置要求，所有组件都能被实例化，而且要能公开其部分细节，并允许你设置和读取它的值。另外，处理 GUI 事件的组件还必须公开其某些方法的细节，这样程序员才能借助 RAD 工具去覆写那些处理事件的方法。Reflection 提供了这样一种机制，它能侦测出组件到底有哪些方法，然后返回这些方法的名字。Java 以 JavaBeans 的形式提供了基于组件的编程的支持。

还有一种“会让你迫切希望能在程序运行时找出类的信息”的情形，这就是通过网络在远程机器上创建对象并运行程序。这被称为“远程方法调用(*Remote Method Invocation* 缩写是 RMI)”。它能让一个 Java 程序将对象分布到很多机器上。这种分布可以有很多原因。比如，你正在进行一项计算密集的任务，为了加快计算速度，你要把任务分拆开来，放到空闲的机器上。或者，你要把处理某些特殊任务的代码(比如在一个多层的客户/服务器架构中，把“业务规则”)放到一台特定的机器上，于是这台机器就成了描述这些行为的共用信息库了，这样系统的任何一个修改就都能很轻易的发生在每个人的身上了。(这是一种很有趣的设计，因为机器只是为了简化软件修改而设的！)除了这几条，分布式计算还能用来发挥一些专用机的特长，有些机器比较擅长一些特殊运算——比如矩阵倒置——而用于通用任务的时候就显得不太合适或太过昂贵了。

除了 **Class** 类(本章前面已经讲过了)，还有一个类库，**java.lang.reflect** 也支持 *reflection*。这个类库里面有 **Field**，**Method**，和 **Constructor** 类(它们都实现了 **Member** 接口)。运行时，JVM 会创建一些这种类的对象来代表未知类的各个成员。然后，你就能用 **Constructor** 来创建新的对象，用 **get( )**和 **set( )**来读取和修改与 **Field** 对象相关联的成员数据，用 **invoke( )**方法调用与 **Method** 对象相关联的方法了。此外，你还能用 **getFields( )**，**getMethods( )**，**getConstructors( )**之类的方法，获取表示成员数据，方法或构造函数的对象数组。(请查阅 JDK 文档以获取更多关于 **Class** 类的细节)由此，即便编译时什么信息都得不到，你也有办法能在运行时问出匿名对象的全部类型信息了。

有一点很重要，reflection 不是什么妖术。当你用 reflection 与未知类的对象打交道的时候，JVM(会和普通的 RTTI 一样)先看看这个对象是属于那个具体类型的，但是此后，它还是得先装载 **Class** 对象才能工作。也就是，不管是从本地还是从网络，反正 JVM 必须拿到那个**.class** 文件。所以 RTTI 同 reflection 的真正区别在于，RTTI 是在编译时让编译器打开并且检查**.class** 文件。换句话说，你是在通过“正常”途径调用对象的方法。而对 reflection 来说，编译时是得不到**.class** 文件的；所以它是在运行时打开并检查那个文件。

一个提取类的方法的程序

www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

一般来说，你不太会直接使用 reflection；Java 之所以要有这种功能，是要用它来支持一些别的特性，比如对象的序列化(第 12 章)和 JavaBeans(第 14 章)。不过在有些情况下，能动态提取类的信息还是很有用的。如果有个能提取类的方法的工具，那就太好了。正如我们前面所讲的，读源代码或是查 JDK 文档的时候，你只能找到这个类的定义里面有的或是覆写的方法。但是这个类还可能从其基类那里继承了很多别的方法。要把它们全都找出来会非常吃力。[1]所幸的是，我们可以用 reflection 写一个小程序，让它自动显示类的完整接口。下面就是它工作的方式：

```
//: c10:ShowMethods.java
// Using reflection to show all the methods of a
class,
// even if the methods are defined in the base class.
// {Args: ShowMethods}
import java.lang.reflect.*;
import java.util.regex.*;

public class ShowMethods {
  private static final String usage =
    "usage: \n" +
    "ShowMethods qualified.class.name\n" +
    "To show all methods in class or: \n" +
    "ShowMethods qualified.class.name word\n" +
    "To search for methods involving 'word'";
  private static Pattern p =
Pattern.compile("\\w+\\.");
  public static void main(String[] args) {
    if(args.length < 1) {
      System.out.println(usage);
      System.exit(0);
    }
    int lines = 0;
    try {
      Class c = Class.forName(args[0]);
      Method[] m = c.getMethods();
      Constructor[] ctor = c.getConstructors();
      if(args.length == 1) {
        for(int i = 0; i < m.length; i++)
          System.out.println(

p.matcher(m[i].toString()).replaceAll(""));
        for(int i = 0; i < ctor.length; i++)
          System.out.println(

p.matcher(ctor[i].toString()).replaceAll(""));
        lines = m.length + ctor.length;
      } else {
        for(int i = 0; i < m.length; i++)
          if(m[i].toString().indexOf(args[1]) != -1)
{
            System.out.println(

p.matcher(m[i].toString()).replaceAll(""));
```

[1]特别是过去。不过 Sun 已经作了很大的改进，所以现在查基类已经简单了很多。

www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

```
            lines++;
          }
        for(int i = 0; i < ctor.length; i++)
          if(ctor[i].toString().indexOf(args[1]) !=
-1) {
            System.out.println(p.matcher(
              ctor[i].toString()).replaceAll(""));
            lines++;
          }
      }
    } catch(ClassNotFoundException e) {
      System.out.println("No such class: " + e);
    }
  }
} ///:~
```

Class 的 **getMethods( )**和 **getConstructors( )**方法分别会返回一个 **Method** 和一个 **Constructor** 数组。这两个类又包括一些“能把它们所代表的方法的名字，参数，返回值全部拆解开来”的方法。不过你也可以像这里所做的，只用 **toString( )**去获取一个包括这个方法的全部特征签名的 **String**。剩下的代码就是用来抽取命令行信息，以及判断方法特征是否与你你输入的字符串相匹配(用 **indexOf( )**)，并且把匹配的方法列出来的。

为了能把“**java.lang**”这样的限定词从“**java.lang.String**”当中抽取出来，我们用了 JDK 1.4 所提供的正则表达式(*regular expression*)。这是一种强大而简洁的工具，有些语言已经用了很长时间了。你已经从 **com.bruceeckel.simpletest.Test** 的 **expect( )**语句中体会到正则表达式的简洁了，而上面那个程序又向你演示了正则表达式的基本编程步骤。

import java.util.regex 之后，你首先用 **static** 的 **Pattern.compile()**方法编译正则表达式。它会返回一个用这个字符串生成的 **Pattern** 对象。这里，这个参数就是

```
"\\w+\\."
```

要想弄懂这个正则表达式，以及其它正则表达式都表示什么意思，请查阅 JDK 文档的 **java.util.regex.Pattern** 部分。对于这个表达式，你要知道‘**\w**’表示“一个单词字符：[a-zA-Z_0-9]”。‘+’的意思是“一个或多个前面的表达式”——所以它表示一个或多个单词字符——以及其后的点，‘\.’(不能只是一个点，在正则表达式里，单独一个点号表示“任意字符”)。所以这个表达式会匹配“一个由单个或多个单词字符再加上一个点号构成”的字符序列，而这正是我们要剥除的。

www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

经过编译获取了 **Pattern** 对象之后，你就可以把待处理的字符串传给它的 **matcher( )**方法了。这个方法会返回一个 **Matcher** 对象，这个对象包括了一套可供你选择的操作(建议你去看看 JDK 文档的 **java.util.regex.Matcher** 部分)。这里只用到了它的 **replaceAll( )** 方法，它会把字符串里所有与表达式相匹配的部分全部替换成空字符——也就是把它们全删了。

还有一种更简单的方法，你可以直接利用 **String** 内置的正则表达式。比如，上面那个程序的 **replaceAll( )**：

```
p.matcher(ctor[i].toString()).replaceAll("")
```

可以改写成

```
ctor[i].toString().replaceAll("\\w+\\.", "")
```

这样就不用预先编译正则表达式了。这种做法很适合那些只用一次的正则表达式，但是如果要反复使用正则表达式，预编译的方式会比较高效。而本例属于后者。

这段程序演示了 reflection 是怎样工作的，由于 **Class.forName( )**所返回的结果在编译时是无法预知的，因此这些方法的特征签名应该是在运行时获取的。如果你研究过 JDK 的文档的话，你就会知道 reflection 的功能之强，足以让你在运行时创建一个“你在编译时一无所知的类”的对象，并调用它的方法(本书的后面会有这个例子的)。尽管开始的时候，你会认为你大概永远也不用不到这些功能，但是 reflection 的价值之高会出乎你的意料。

提示一下这个程序该怎样运行

```
java ShowMethods ShowMethods
```

这样就能得到 **ShowMethods** 类的所有方法了。尽管源代码里没定义构造函数，但是清单里还是有一个 **public** 的默认构造函数。这是编译器自动合成的。如果你把这个类改成非 **public** 的(也就是 package 权限的)，那么这个合成的默认构造函数就不会有了。合成出来的默认构造函数的访问权限会和类的相同。

还有一个有趣的实验，试试用 **char**，**int**，**String** 之类的参数去调用 **java ShowMethods java.lang.String**。

www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

这个工具能帮你省下很多时间。编程的时候，如果你记不起来类有没有这个方法，或者能不能这样使用，比方说 **Color** 对象的时候，就再也不用着去翻箱倒柜地去查 JDK 文档了。

第 14 章还有一个这个程序的 GUI 版(专为提取 Swing 组件的信息而定制的)，所以你编程的时候可以把这个程序打开，这样查起来会更快些。

总结

RTTI 能让你用一个匿名的基类 reference 来获取对象的确切类型的信息。在不懂多态方法调用的时候，这么做是理所当然的，因此新手们会自然而然的想到它，于是就用错了地方。对很多从面向过程的编程语言转过来的人来说，刚开始的时候，他们还不习惯扔掉 **switch** 语句。于是当他们用 RTTI 来编程的时候，就会错过多态性所带来的编程和代码维护方面的好处。Java 的本义是让你在程序里面全程使用多态性，只是在万不得已的情况下才使用 RTTI。

但是要想正确地使用多态方法调用，你就必须要能控制基类的定义。因为当你扩展程序的时候，可能会发现基类里面没有你想要的方法。如果这个基类是来自类库的，或是由别人控制的，那么 RTTI 就成解决方案了：你可以继承一个新的类，然后加上你自己的方法。在程序的其它地方，你可以检测出这个类型，调用那些特殊的方法。这样做不会破坏多态性，也不影响程序的扩展性，因为加一个新的类型不会要你去到处修改 **switch** 语句。但是，如果是在程序的主体部分加入要使用新特性的代码的话，你就必须使用 RTTI 来检查对象的确切类型了。

把特性加入基类就意味着为了迁就某个类，所有其它从这个基类继承下来的类都得加上一些毫无意义的方法。这就使接口变得很不清晰了，而且继承的时候，还要把这些 abstract 的方法全都填满，这是件很烦人的事。比方说，有一个表示乐器的类系。假设你要把乐团里的所有乐器的喇叭嘴全都清洗一遍，把 **clearSpitValve( )**方法放进 **Instrument** 基类算一个办法，但是这就等于在说 **Percussion**(打击乐器)和 **Electronic** 乐器也有喇叭嘴，所以这种做法是很浆糊的。在这种情况下，RTTI 提供了一种更为合理的解决方案，因为你可以把这个方法放到一个合适的子类里面(这里就是 **Wind**)。但是更好的办法应该是往基类里放一个 **prepareInstrument( )**方法。不过第一次碰到这类问题的时候，一般不会去这么想，你会误认为只能使用 RTTI。

最后，RTTI 还会被用来解决效率问题。假设你写了一个很好的多态程序，但是运行的时候发现，有个对象反映奇慢。于是，你就可以用 RTTI 把这个对象拣出来，然后专门针对它的问题写代码以提高程序的运行效率。不过编程的时候切忌去过早优化代码。这是一个很有诱惑的陷阱。最好还是先让程序跑起来，然后再判断一下它跑得是不是够快了。只有觉得

www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

它还不够快，你才应该去着手解决效率问题——用 profiler(见第 15 章)。

练习

只要付很小一笔费用就能从 *www.BruceEckel.com* 下载名为 *The Thinking in Java Annotated Solution Guide* 的电子文档，这上面有一些习题的答案。

1. 往 **Shapes.java** 里面加一个 **Rhomboid** 类。创建一个 **Rhomboid** 对象，上传给 **Shape**，然后再下传回 **Rhomboid**。试着把它下传给 **Circle**，看看会有什么结果。
2. 修改练习 1，在下传前用 **instanceof** 检查一下对象的类型。
3. 修改 **Shapes.java**，让它能“突出”(设置一个标记)某种形状。**Shape** 的各个派生类的 **toString( )**方法要能标识这个 **Shape** 是不是被“突出”出来了。
4. 修改 **SweetShop.java**，用命令行参数去控制各种对象的创建。也就是说，如果命令行是“**java SweetShop Candy**”，那么它只创建 **Candy** 对象。观察一下，你是怎样通过命令行参数来控制 **Class** 对象的装载的。
5. 往 **PetCount3.java** 里面加一个新的 **Pet** 类。验证一下，**main( )**能正确地创建并且计算这类对象。
6. 写一个方法，这个方法要能接受一个对象，然后递归打印这个对象所属类系的所有类。
7. 修改练习 6，让它用 **Class.getDeclaredFields( )**方法把类的数据成员也给打印出来。
8. 找到 **ToyTest.java**，把 **Toy** 的默认构造函数注释掉，然后说明一下，会有什么结果。
9. 往 **ToyTest.java** 里面加一个 **interface**，验证一下，它会被正确地检测并且显示出来。
10. 写一个程序让它判断一下，**char** 数组究竟是一种 **primitive** 还是一个货真价实的对象。
11. 按照“总结”部分所说的，实现 **clearSpitValve( )**方法。
12. 按照本章所述，实现 **rotate(Shape)**方法，让它检查一下，要旋转的形状是不是 **Circle**(如果是，就不进行这项操作)。
13. 找到 **ToyTest.java**，利用 reflection 机制，调用非默认构造函数创建一个 **Toy** 对象。

www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

14. 查阅 JDK 文档的 **java.lang.Class** 部分。写一个程序，让它用 Class 方法，把用命令行参数给出的类的所有信息全都提取出来。用标准类库里的类，以及你自创的类做检验。

15. 修改 **ShowMethods.java** 中的正则表达式，让它再剥离 **native** 和 **final** 这两个关键词(提示：使用“或”运算符‘|’。)

www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com