Reflection

Ability of a program to discover information about objects and their classes at runtime.

Also called

Run-time Type Information (RTTI)

Polymorphism (dynamic binding of methods) and downcasting are a form of basic RTTI: The type of an object must be identified at runtime.

The operation **instanceof** tests the type of an object.

The Class Class

Instances of the class Class, a subclass of Object found in the *java.lang* package, represent the types in Java, namely classes, interfaces, arrays, and primitive types.

Class has no public constructor.

Creating Class Objects

- An instance method in Object

  ob.getClass()   returns a Class object

- Methods in Class

  Class.forName("Domino")   returns a Class object

  May throw ClassNotFoundException,
  a checked exception.

```
Domino d = new Domino();
Class dc = d.getClass();
Class sc = dc.getSuperclass();
```

- Class constants

If T is any Java type,

T.class is the corresponding Class object.

Constants for primitive types and constants in wrapper classes:

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

Using Class Objects

```
Class cd = Class.forName("Domino");

String n  = cd.getName();         // the String "Domino"
```

For any Object ob,

ob.getClass().getName()

returns the name of ob’s class as a String.

Provided its class has a no-argument constructor,

ob.getClass().newInstance()

returns a new instance of that class.

cd.newInstance()   returns a new default Domino object.

Notes

- *newInstance* may throw the checked exceptions InstantiationException and IllegalAccessException.
- *newInstance* returns the type Object, which may need to be downcast.
- Boolean.TYPE is the same kind of object as boolean.class, but both are different from Boolean.class.

Properties of a Class Object

Suppose *cob* refers to a Class object.

Method Call	**Return value**
cob.getSuperclass()	a Class object
cob.getInterfaces()	an array of Class objects
cob.isInterface()	boolean
cob.isArray()	boolean
cob.isPrimitive()	boolean
cob.getFields()	array of Field (public ones, including inherited)
cob.getDeclaredFields()	array of Field (all local ones)
cob.getMethods()	array of Method
cob.getDeclaredMethods()	array of Method
cob.getConstructors()	array of Constructor
cob.getDeclaredConstructors()	"
cob.getModifiers()	an **int** that encodes modifiers

Use methods in class Modifier to decode the **int**.

Modifier Coding

1	public
2	private
4	protected
8	static
16	final
32	synchronized
64	volatile
128	transient
256	native
512	interface
1024	abstract

The Modifier class has **boolean** class methods:

Modifier.isPublic(**int**)
Modifier.isPrivate(**int**)
:
Modifier.isStatic(**int**)
:
Modifier.isAbstract(**int**)

and

Modifier.toString(**int**)
returns a String listing the modifiers.

Package java.lang.reflect

Classes

Field
Method
Constructor
Modifier
Array

Creating a new Object with constructor parameters:

- Get an instance of Constructor

```
Class cd = Class.forName("Domino");

Class [] params = { int.class, int.class, boolean.class };

Constructor con = cd.getConstructor(params);
```

Note: Constructor must be public.

- Construct an array of actual parameters

```
Object [] aparams =
            { new Integer(4), new Integer(7),
                              new Boolean(true) };
```

- Create a new instance with the actual parameters

```
Object newd = con.newInstance(aparams);

System.out.println("newd = " + newd);
      newd = <4, 7> UP
```

Where downcasting is necessary

```
   int high = ((Domino)newd).getHigh();
```

Discovering the Nature of a Class

The methods in Field, Method, and Constructor allow us to determine the syntactic properties inside of a class.

Notice how the various parts of a class are extracted and printed in the program Discover.java.

Although we can extract the types of instance and class variables and the signatures of constructors and instance and class methods, we cannot inspect the code inside methods or in a static initializer.

Since inner classes are purely a compile-time device, we cannot find evidence of them in the class under investigation.

Discover.java

```
import java.lang.reflect.*;
import java.util.Scanner;

public class Discover
{
   public static void main(String [] args)
   {
      try
      {
         Scanner scan = new Scanner(System.in);
         System.out.print("Enter a class or interface name "
                                             + " (e.g. java.util.Date): ");
         String name = scan.nextLine();
         Class cl = Class.forName(name);
         int mods = cl.getModifiers();
      if (mods>0)
        System.out.print(Modifier.toString(mods)+" ");

      if (!cl.isInterface())              // interface is a modifier
            System.out.print("class ");
      System.out.print(cl.getName());
      Class supercl = cl.getSuperclass();
      if (supercl != null &&!supercl.equals(Object.class))
          System.out.print(" extends " + supercl.getName());
      Class [] interfaces = cl.getInterfaces();
      if (interfaces.length>0)
          System.out.print(" implements ");
      for (int k=0; k<interfaces.length; k++)
      {
        if (k>0) System.out.print(", ");
        System.out.print(interfaces[k].getName());
      }
      System.out.print("\n{\n");

      printConstructors(cl);
      System.out.println();

      printMethods(cl);
      System.out.println();

      printFields(cl);
      System.out.println("}");
    }
    catch (ClassNotFoundException e)
    { System.out.println("Class not found"); }
  }
```

```
public static void printConstructors(Class cl)
{
   Constructor [] constructors = cl.getDeclaredConstructors();

   for (int c = 0; c < constructors.length; c++)
   {
      Constructor con = constructors[c];
      System.out.print("     ");

      int mods = con.getModifiers();
      if (mods>0)
         System.out.print(Modifier.toString(mods)+" ");

      System.out.print(cl.getName() + "(");

      Class [] paramTypes = con.getParameterTypes();
      for (int k=0; k<paramTypes.length; k++)
      {
         if (k>0) System.out.print(", ");
         Class param = paramTypes[k];
         if (param.isArray())
            System.out.print(
               param.getComponentType().getName()+" []");
         else
            System.out.print(param.getName());
      }
      System.out.print(")");

      Class [] excepts = con.getExceptionTypes();
      if (excepts.length>0)
                           System.out.print(" throws ");
      for (int k=0; k<excepts.length; k++)
      {
         if (k>0) System.out.print(", ");
         System.out.print(excepts[k].getName());
      }
      System.out.println("; ");
   }
}

public static void printMethods(Class cl)
{
   Method [] methods = cl.getDeclaredMethods();

   for (int m = 0; m < methods.length; m++)
   {
      Method meth = methods[m];
      System.out.print("     ");

      int mods = meth.getModifiers();
      if (mods>0)
         System.out.print(Modifier.toString(mods)+" ");

      Class retType = meth.getReturnType();
      if (retType.isArray())
         System.out.print(
            retType.getComponentType().getName()+" []");
      else
         System.out.print(retType.getName());
      System.out.print(" " + meth.getName() + "(");
      Class [] paramTypes = meth.getParameterTypes();
      for (int k = 0; k < paramTypes.length; k++)
      {
         if (k > 0) System.out.print(", ");
         Class param = paramTypes[k];
         if (param.isArray())
            System.out.print(
               param.getComponentType().getName()+" []");
         else
            System.out.print(param.getName());
      }
      System.out.print(")");

      Class [] excepts = meth.getExceptionTypes();
      if (excepts.length>0)
                           System.out.print(" throws ");
      for (int k=0; k<excepts.length; k++)
      {
         if (k>0) System.out.print(", ");
         System.out.print(excepts[k].getName());
      }
      System.out.println("; ");
   }
}

public static void printFields(Class cl)
{
   Field [] fields = cl.getDeclaredFields();

   for (int f = 0; f < fields.length; f++)
   {
      Field field = fields[f];
      System.out.print("    ");
      int mods = field.getModifiers();
      if (mods>0)
         System.out.print(Modifier.toString(mods)+" ");

      Class type = field.getType();
      if (type.isArray())
         System.out.print(
                  type.getComponentType().getName(+" []");
      else
```

```
        System.out.print(type.getName());
      System.out.println(" " + field.getName() + ";");
    }
  }
}
```

Sample Execution

```
% java Discover
Enter a class or interface name
                (e.g. java.util.Date): domino.Domino

class domino.Domino implements java.io.Serializable
{
   public domino.Domino(int, int, boolean)
         throws java.lang.RuntimeException;
   public domino.Domino() ;

   int getHigh();
   int getLow()  throws java.lang.ClassCastException,
                     java.lang.IllegalArgumentException;
   public java.lang.String toString();
   boolean matches(domino.Domino);
   static int getNumber();

   int spots1;
   int spots2;
   boolean faceUp;
   static final int MAXSPOTS;
   static int numDominoes;
}
```

Notes

cl.getSuperclass() returns **null** if *cl* is

- Object.class
- an interface Class object (even if interface extends another interface—that is viewed as implementing the other interface)
- a Class object of a primitive type

An Aside: Array Literals

An array literal can be written using braces:

```
        { 1, 2, 3, 4 }

        { "mon", "tues", "wed", "thur", "fri" }

        { new Domino(1,2,true), new Domino(2,2,true) }
```

These literal can only be used for initialization of a freshly declared variable:

```
        int [] a = { 1, 2, 3, 4 };
```

Such a literal may not be assigned to an already existing variable:

```
        a = { 2, 4, 5 };    // illegal
```

However, Java does have a way of constructing such array objects that can be assigned dynamically:

```
    a = new int [] { 2, 4, 6 };

    Number [] na;

    na = new Number [] { new Byte((byte)26),

                                    new Short((short)5), new Float(8.8) };
```

Manipulating Fields

Suppose

fd is an object of type Field for some class

and

ob is an object of that class.

Then

fd.get(ob) returns an Object whose value is the current value of the field *fd* in *ob*.

For primitive types:

fd.get(ob) returns the value wrapped as an object of corresponding type: Integer for **int**, etc.

Alternatively, use special *getX* methods:

fd.getBoolean(ob)	returns **boolean**
:	:
fd.getInt(ob)	returns **int**
:	:
fd.getDouble(ob)	returns **double**

Condition: Need field *fd* to be visible by one of these means.

- field *fd* is **public**
- *get* call is inside the same class (**private**)
- *get* call is inside a subclass (**protected**)

Setting Field Values

```
        fd.set(ob, value);
```

where *value* is an Object of the appropriate type.

```
        void set(Object o, Object v)
```

Also have individual methods:

```
        fd.setBoolean(ob, b)
                        :
        fd.setChar(ob, c)
                        :
        fd.setLong(ob,g)
```

Example: TestFields.java

The class InspectFields has two methods:

1. *printFields()* displays information about each field that is visible.

2. *changeField(String f, Object val)* changes the value of the field *f* to the object *val*.

The main class, TestFields, creates an object, prints its fields, changes the values of some of its fields, and then prints the fields again.

Class To Be Inspected

These classes and interfaces are saved in three different files.

```
import java.util.Date;
public class B extends A implements InFace
{
    public Date myDate;
    public double myDouble;
    float myFloat;
    public short myShort;
    public static byte myStatic;

    public B()
    {
        myDouble = 3.14;
        myFloat = (float)1.14;      // myFloat = 1.14F;
        myShort = 1492;
        myDate = new Date();
    }

    public int bar()
    {  return 0;  }
}

public interface InFace
{
    long LONG = 1000;
    int bar();
}
```

```
public class A
{
      protected int myInt;
      public String str;

      public A()
      {
            myInt = 2000;
            str = "Hello";
      }

      public void foo(int i)
      { }
}
```

Code for TestFields

```
import java.lang.reflect.*;
import java.util.Date;

public class TestFields
{
   public static void main(String [] args)
   {
      B b = new B();

      InspectFields insFlds = new InspectFields(b);

      insFlds.printFields();

      insFlds.changeField("LONG", new Long(123456789000L));
      insFlds.changeField("str", "Goodbye");
      insFlds.changeField("myInt", new Integer(119));
      insFlds.changeField("myDate", new Date());
      insFlds.changeField("myFloat", new Float(6.66));
      insFlds.changeField("mydouble", new Double(12.34));
      insFlds.changeField("myShort", new Long(96));
      insFlds.changeField("myShort", new Short((short)2001));
      insFlds.changeField("myStatic", new Byte((byte)-99));
      insFlds.printFields();
   }
}
```

Code for InspectFields

```
class InspectFields
{
   Object myObj;

   InspectFields(Object obj)
   {  myObj = obj;  }

   void printFields()
   {
      try
      {
         Class cl = myObj.getClass();
         Field [] fields = cl.getFields();      // public fields only
         for (int f = 0; f < fields.length; f++)
         {
            Field field = fields[f];
            System.out.println("Name: " + field.getName());
            System.out.println("Declaring class: "
                                + field.getDeclaringClass());
            int mods = field.getModifiers();
            System.out.println("Modifiers: "
                                + Modifier.toString(mods));
            System.out.println("Type: " + field.getType());
            System.out.println("Declaration: " + field.toString());
            System.out.println("Value: " + field.get(myObj));
            System.out.println();
         }
      }
      catch (SecurityException e)
      { System.out.println(e); }
      catch (IllegalAccessException e)
      { System.out.println(e); }
   }

   void changeField(String name, Object val)
   {
      try
      {
            Class cl = myObj.getClass();
            Field field = cl.getField(name);
            field.set(myObj, val);
      }
      catch (SecurityException e)          // Possible only if a security
      { System.out.println(">>>" + e); }   // manager is present.
      catch (NullPointerException e)
      { System.out.println(">>>" + e); }
      catch (IllegalArgumentException e)
      { System.out.println(">>>" + e); }
      catch (IllegalAccessException e)
      { System.out.println(">>>" + e); }
      catch (NoSuchFieldException e)
      { System.out.println(">>>" + e); }
   }
}
```

Output

```
% java TestFields
Name: LONG
Declaring class: interface InFace
Modifiers: public static final
Type: long
Declaration: public static final long InFace.LONG
Value: 1000

Name: str
Declaring class: class A
Modifiers: public
Type: class java.lang.String
Declaration: public java.lang.String A.str
Value: Hello

Name: myDate
Declaring class: class B
Modifiers: public
Type: class java.util.Date
Declaration: public java.util.Date B.myDate
Value: Fri Aug 04 09:24:23 CDT 2000

Name: myDouble
Declaring class: class B
Modifiers: public
Type: double
Declaration: public double B.myDouble
Value: 3.14

Name: myShort
Declaring class: class B
```

```
Modifiers: public
Type: short
Declaration: public short B.myShort
Value: 1492

Name: myStatic
Declaring class: class B
Modifiers: public static
Type: byte
Declaration: public static byte B.myStatic
Value: 0

>>>java.lang.IllegalAccessException: Field is final
>>>java.lang.NoSuchFieldException: myInt
>>>java.lang.NoSuchFieldException: myFloat
>>>java.lang.NoSuchFieldException: mydouble
>>>java.lang.IllegalArgumentException: field type mismatch

Name: LONG
Declaring class: interface InFace
Modifiers: public static final
Type: long
Declaration: public static final long InFace.LONG
Value: 1000

Name: str
Declaring class: class A
Modifiers: public
Type: class java.lang.String
Declaration: public java.lang.String A.str
Value:  Goodbye
```

```
Name: myDate
Declaring class: class B
Modifiers: public
Type: class java.util.Date
Declaration: public java.util.Date B.myDate
Value: Fri Aug 04 09:24:23 CDT 2000

Name: myDouble
Declaring class: class B
Modifiers: public
Type: double
Declaration: public double B.myDouble
Value: 3.14

Name: myShort
Declaring class: class B
Modifiers: public
Type: short
Declaration: public short B.myShort
Value: 2001

Name: myStatic
Declaring class: class B
Modifiers: public static
Type: byte
Declaration: public static byte B.myStatic
Value: -99
```

Dynamic Array Creation

Create an array at runtime whose

- component type is determined dynamically
- length is determined dynamically

Class Instance Method for Arrays

Class getComponentType()

Array Class Methods

static int getLength(Object array)

static Object newInstance(Class cType, **int** len)

Also can get and set components in an array in a manner similar to the fields in an object.

static Object get(Object array, **int** index)

static void set(Object array, **int** index, Object val)

For *array* we need a return type for *get* and parameter type for *set* that is a superclass of both Object [] and **int** [].

Application

- Given an array, create another array with the same components but whose length is double.
- Print the array type and its components.

Example: ArrayGrow

Main method creates three arrays, grows them, and prints the new arrays.

Note that the array objects need to be downcast.

```
import java.lang.reflect.Array;
public class ArrayGrow
{
    public static void main(String [] args)
    {
        int [] a = { 1, 2, 3, 4, 5, 6 };
        a = (int [])arrayGrow(a);
        arrayPrint(a);

        String [] b = { "one", "two", "three" };
        b = (String [])arrayGrow(b);
        arrayPrint(b);
        Integer [] c = { new Integer(1), new Integer(2) };
        c = (Integer [])arrayGrow(c);
        arrayPrint(c);
    }

    static Object arrayGrow(Object a)
    {
        Class cl = a.getClass();
        if (!cl.isArray()) return null;
        Class ct = cl.getComponentType();
        int len = Array.getLength(a);

        // Create new array instance with double length
        Object newArray = Array.newInstance(ct, 2*len);
        // Copy old array into new
        System.arraycopy(a, 0, newArray, 0, len);
        return newArray;
    }

    static void arrayPrint(Object a)
    {
        Class cl = a.getClass();
        if (!cl.isArray()) return;
        Class ct = cl.getComponentType();
        int len = Array.getLength(a);
        System.out.println("\n" + ct.getName()
                                            + "[" + len + "]");
        for (int k = 0; k < len; k++)
            System.out.println(Array.get(a, k));        // ≡ a[k]
    }
}
```

Output

```
int[12]      java.lang.String[6]     java.lang.Integer[4]
1            one                     1
2            two                     2
3            three                   null
4            null                    null
5            null
6            null
0
0
0
0
0
0
```

Method Objects

In class Class, an instance method:

```
public Method getMethod(String name, Class [] paramTypes)
                    throws NoSuchMethodException,
                           SecurityException
```

In class Method, an instance method:

```
public Object invoke(Object obj, Object [] args)
                    throws IllegalAccessException,
                           IllegalArgumentException,
                           InvocationTargetException
```

For a class method, *obj* is ignored and may be **null**.

For an instance method, *obj* refers to the object that is executing the method, the receiver.

Functional Programming

Functions (methods) are first-class entities:

1. can be stored in data structures
2. can be passed as parameters
3. can be returned as function results
4. can be defined by a literal expression

Java Methods

- Using reflection mechanisms, we can provide features 1 and 2.
- We can return a method as the result of a function, but there is no way to construct functions (methods) dynamically.

Consider the problem of defining a method that composes two functions, $f \circ g$:

Method compose(Method f, Method g)

This method cannot be written in Java.

Higher-order Functions

- Map (apply-to-all)

  map(f, {1, 2, 3}) = { f(1), f(2), f(3) }

- Construction

  construct({ f_1, f_2, f_3 }, x) = { f_1(x), f_2(x), f_3(x) }

- Filter

  filter(f, { a, b, c }) = { x∈{ a, b, c } | f(x) = true }

- Reduce (fold-left)

  reduce(f, z, {a,b,c}) = f(f(f(z, a), b), c)

Implementing Map

Create three objects for methods with signature double → double.

Pass each to a *map* method along with an array of double values.

```
import java.lang.reflect.Method;
import java.text.NumberFormat;
import java.text.DecimalFormat;

public class Map
{
   public static void main(String [] args) throws Exception
   {
      double [] list = { 12.5, 20.2, -94.66, 802.6 };
      double [] newList;
```

```
    printList(list);

    Method f1 = Map.class.getMethod("sqr",
                          new Class [] { double.class });
    System.out.println("\n" + f1);
    newList = map(f1, list);
    printList(newList);

    Method f2 = Math.class.getMethod("rint",
                          new Class [] { double.class });
    System.out.println("\n" + f2);
    newList = map(f2, list);
    printList(newList);

    Method f3 = Map.class.getMethod("recip",
                          new Class [] { double.class });
    System.out.println("\n" + f3);
    newList = map(f3, list);
    printList(newList);
}

public static double sqr(double x) { return x*x; }

public static double recip(double x) { return 1/x; }

public static double [] map(Method f, double [] list)
{
    double [] result = new double [list.length];
    for (int k = 0; k < list.length; k++)
    {
        try
        {
            Object [] args = { list[k] };
            result[k] = (Double)f.invoke(null, args);
        }
```

```
          catch (Exception e)
          { System.out.println("???"); }
      }
      return result;
   }

   public static void printList(double [] list)
   {
      for (int k = 0; k < list.length; k++)
            System.out.print(format(list[k]));
      System.out.println();
   }

   static String format(double b)
   {
      NumberFormat nf =
         new DecimalFormat("###0.0000;-###0.0000");
      String val = nf.format(b);
      return "            ".substring(0,12-val.length())+val;
   }
}
```

Output

```
12.5000     20.2000    -94.6600    802.6000

public static double Map.sqr(double)
    156.2500    408.0400   8960.5156 644166.7600

public static native double java.lang.Math.rint(double)
     12.0000     20.0000    -95.0000    803.0000

public static double Map.recip(double)
      0.0800      0.0495     -0.0106      0.0012
```