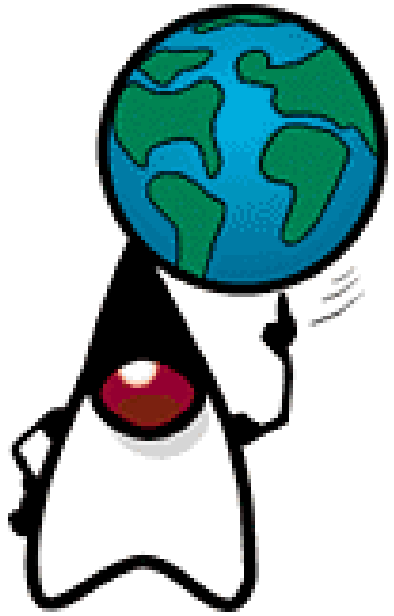
Using Java™ Reflection



What Is Reflection ?

- Java™ Technology provides two ways to discover information about an object at runtime
  - Traditional runtime class identification
    - The object's class is available at compile and runtime
    - Most commonly used
  - Reflection
    - The object's class may not be available at compile or runtime

- "Reflection in a programming language context refers to the ability to observe and/or manipulate the inner workings of the environment programmatically."[1]

- "The reflection API represents, or reflects, the classes, interfaces, and objects in the current Java™ virtual machine."[2]

1. J. R. Jackson, A. L. McClellan, Java™ 1.2 By Example, Sun Microsystems, 1999.
2. M. Campione, et al, The Java™ Tutorial Continued, Addison Wesley, 1999.

Why Runtime Class Identification ?

- There is a class named `Class`, instances of which contain runtime class definitions
  - In a C or C++ program, you have a pointer to an object but you don't know what type of object it is, there is no way to find out
  - In Java, finding out based on the runtime type information is straightforward

- Java™ technology takes advantage of polymorphism
  - New subclasses easily added
  - Bulk of behaviors inherited from its superclass
  - No impact on other subclasses of the superclass
  - At runtime, the JVM™ takes advantage of late dynamic binding
  - Messages are directed to the correct method

How the Class Object Works?

- Every class loaded into the JVM™ has a `Class` object
  - Corresponds to a *`.class`* `file`
  - The `ClassLoader` is responsible for finding and loading the class into the JVM™
- At object instantiation
  - The JVM™ checks to see if the class is already loaded into the virtual machine
  - Locates and loads the class on an as needed basis
  - Once loaded, the JVM™ uses the loaded class to instantiate an instance
- Example: If you know the name of the class at compile time, you can retrieve its `Class` object by appending *`.class`* to its name:

```
Class c = java.awt.Button.class;
```

Late Dynamic Binding

- The J2SE Java Runtime Environment (JRE) does not require that all classes are loaded prior to execution
  - Different from most other environments
- Class loading occurs when the class is first referenced
- Late Dynamic Binding is…
  - Important for polymorphism
    - Message propagation is dictated at runtime
    - Messages are directed to the correct method
  - Essential for reflection to be possible

Class Literals

- Using `Class` Literals is the second way to reference an object's class
- In Java™ the class "`java.lang.Class`" has always been used to represent class and interface types. A statement like:

```
Class c = null;
try {
     c = Class.forName("java.lang.String");
}
catch (Throwable e) {
}
```

  will dynamically load the specified class if it is not already loaded
- In JDK™ 1.0 a `Class` object can also represent arrays, as a special type of class
- In JDK™ 1.1, this notion has been broadened to include any Java type, and there is an easier way to obtain the `Class` object for a type
  - All classes, interfaces, arrays, and primitive types have class literals
  - Primitive types have corresponding wrapper classes

Class Literals

- For example, the following code prints "equal":

```
public class test1 {

  public static void main(String args[]) {
     try {
          Class c1 = Class.forName("java.lang.String");
          Class c2 = java.lang.String.class;
          if (c1 == c2)
              System.out.println("equal");
        }
      catch (Throwable e) {
        }
     }
}
```

Class Literals

- This feature can also be used on fundamental types, for example:

```
public class test2 {

  public static void main(String args[]) {
         Class c1 = Integer.TYPE;
         Class c2 = int.class;
         if (c1 == c2)
            System.out.println("equal");
       }
}
```

- `Integer.Type` returns the `Class` instance representing the primitive type `int`

The `instanceof` Keyword

- The `instanceof` keyword is the third way to reference an object's class
  - Used with both classes and interfaces
- Returns true if the object is a species of a specified class
  - Subclasses will also answer true
  - Code becomes structurally bound to the class hierarchy
- Several limitations on the referenced class
  - Must be a named class or interface
  - The class constant cannot be the `Class` class
- Example:

```
if (x instanceof Circle)
   ((Circle) x).setRadius (10);
```

The Reflection API

- The reflection API is the fourth way to reference an object's class
- Reflection allows programs to interrogate and manipulate objects at runtime
- The reflected class may be…
  - Unknown at compile time
  - Dynamically loaded at runtime

Core Reflection Classes

- `java.lang.reflect`
  - The reflection package
  - Introduced in JDK 1.1 release
- `java.lang.reflect.AccessibleObject`
  - The superclass for **`Field`**, **`Method`**, and **`Constructor`** classes
  - Suppresses the default Java language access control checks
  - Introduced in JDK 1.2 release
- `java.lang.reflect.Array`
  - Provides static methods to dynamically create and access Java arrays
- `java.lang.reflect.Constructor`
  - Provides information about, and access to, a single constructor for a class

Core Reflection Classes (Cont.)

- `java.lang.reflect.Field`
  - Provides information about, and dynamic access to, a single field of a class or an interface
  - The reflected field may be a class (static) field or an instance field
- `java.lang.reflect.Member`
  - Interface that reflects identifying information about a single member (a field or a method) or a constructor
- `java.lang.reflect.Method`
  - Provides information about, and access to, a single method on a class or interface
- `java.lang.reflect.Modifier`
  - Provides static methods and constants to decode class and member access modifiers

Core Reflection Classes (Cont.)

- JDK 1.3 release additions
  - `java.lang.reflect.Proxy`
    - Provides static methods for creating dynamic proxy classes and instances
    - The superclass of all dynamic proxy classes created by those methods
  - `java.lang.reflect.InvocationHandler`
    - Interface implemented by the invocation handler of a proxy instance

Commonly Used Classes

- `java.lang.Class`
  - Represents classes and interfaces within a running Java™ technology-based program
- `java.lang.Package`
  - Provides information about a package that can be used to reflect upon a class or interface
- `java.lang.ClassLoader`
  - An abstract class
  - Provides class loader services: Three types
    - The bootstrap class loader
    - The extension class loader
    - The application class loader

Design Strategies for Using Reflection

- Challenge switch/case and cascading if statements
  - Rationale
    - The switch statement should scream “redesign me” to the developer
    - In most cases, switch statements perform pseudo subclass operations
  - Steps
    - Redesign using an appropriate class decomposition
    - Eliminate the switch/case statement
    - Consider a design pattern approach
  - Benefits
    - High level of object decoupling
    - Reduced level of maintenance

Example: Shape Factory Without Reflection

```
public static Shape getFactoryShape (String s)
    {
     Shape temp = null;
     if (s.equals (“Circle”))
        temp = new Circle ();
       else
        if (s.equals (“Square”))
           temp = new Square ();
          else
           if (s.equals (“Triangle”)
                 temp = new Triangle ();
             else
               // …
            // continues for each kind of shape
      return temp;
    }
```

Example: Shape Factory With Reflection

```
public static Shape getFactoryShape (String s)
   {
     Shape temp = null;
      try
      {
       temp = (Shape) Class.forName (s).newInstance ();
      }
     catch (Exception e)
      {
      }
     return temp;
   }
```

Using Reflection

- Reflection allows programs to interrogate an object at runtime without knowing the object's class
- How can this be…
  - Connecting to a JavaBean™ technology- based component
  - Object is not local
    - RMI or serialized object
- Reflection solves problems within object-oriented design:
  - Flexibility
  - Extensibility
  - Pluggability
- Reflection solves problems caused by…
  - The static nature of the class hierarchy
  - The complexities of strong typing

What Can I Do With Reflection

- Literally everything that you can do if you know the object's class
  - Load a class
  - Determine if it is a class or interface
  - Determine its superclass and implemented interfaces
  - Instantiate a new instance of a class
  - Determine class and instance methods
  - Invoke class and instance methods
  - Determine and possibly manipulate fields
  - Determine the modifiers for fields, methods, classes, and interfaces
  - Working with Arrays
  - Etc.

Here Is How To…

- Load a class

```
Class c = Class.forName (“Classname”)
```

- Determine if a class or interface

```
c.isInterface ()
```

- Determine lineage
  - Superclass

```
Class c1 = c.getSuperclass ()
```

  - Superinterface

```
Class[] c2 = c.getInterfaces ()
```

Here Is How To…

- Determine implemented interfaces

```
Class[] c2 = c.getInterfaces ()
```

- Determine constructors

```
Constructor[] c0 = c.getDeclaredConstructors ()
```

- Create an instance
  - Default constructor

```
Object o1 = c.newInstance ()
```

  - Non-default constructor

```
Constructor c1 = c.getConstructor (class[]{…})
Object i = c1.newInstance (Object[] {…})
```

Here Is How To…

- Determine methods

```
Methods[] m1 = c.getDeclaredMethods ()
```

- Find a specific method

```
Method m = c.getMethod (“methodName”,
                        new Class[] {…})
```

- Invoke a method

```
m.invoke (c, new Object[] {…})
```

Here Is How To…

- Determine modifiers

```
Modifiers[] mo = c.getModifiers ()
```

- Determine fields

```
Class[] f = c.getDeclaredFields ()
```

- Find a specific field

```
Field f = c.getField()
```

- Modify a specific field
  - Get the value of a specific field

```
f.get (o)
```

  - Set the value of a specific field

```
f.set (o, value)
```

An Example

```
import java.lang.reflect.*;
import java.awt.*;

class SampleField {

   public static void main(String[] args) {
        Foo f = new Foo();
        printFieldNames(f);
       }
   // …
}
```

Continuing the Example

```
static void printFieldNames(Object o) {
  Class c = o.getClass();
  Field[] publicFields = c.getFields();
  for (int i = 0; i < publicFields.length; i++) {
      String fieldName = publicFields[i].getName();
      Class typeClass = publicFields[i].getType();
      String fieldType = typeClass.getName();
      System.out.println("Name: " + fieldName +
                         ", Type: " + fieldType);
    }
}
```

The Field Class

- Getting the value associated with a `Field`
  - `Object get(Object obj):` Gets the value of a field as a an object
  - `short getShort(Object obj):` Gets the value of a field as a short on the specified object
  - `Boolean getBoolean(Object obj):` Gets the value of a field as a boolean on the specified object
  - And so on – each specific type `get` method throws a `IllegalAccessException` and a `NoSuchFieldException`

An Example

```
import java.lang.reflect.*;
import java.awt.*;

class SampleGet {

   public static void main(String[] args) {
      Rectangle r = new Rectangle(100, 325);
      printHeight(r);
   }
// ..
```

Continuing the Example

```
static void printHeight(Rectangle r) {
 Field heightField;
 Integer heightValue;
 Class c = r.getClass();
 try {
      heightField = c.getField("height");
      heightValue = (Integer) heightField.get(r);
      System.out.println("Height: " +
                         heightValue.toString());
    } catch (NoSuchFieldException e) {
           System.out.println(e);
    } catch (SecurityException e) {
           System.out.println(e);
    } catch (IllegalAccessException e) {
           System.out.println(e);
    }
}
```

Manipulating Objects

- Software development tools, such as GUI builders and debuggers, need to manipulate objects at runtime
- For example, a GUI builder may allow the end-user to select a `Button` from a menu of components, create the `Button` object, and then click the `Button` while running the application within the GUI builder

Setting Field Values

- To modify the value of a field you have to:
  - Create a `Class` object
  - Create a `Field` object by invoking `getField` on the `Class`
  - Invoke the appropriate `set` method on the `Field` object

- The `Field` class provides several `set` methods
  - Specialized methods, such as `setBoolean` and `setInt`, are for modifying primitive types
  - If the field you want to change is an object invoke the `set` method. You can call `set` to modify a primitive type, but you must use the appropriate wrapper object for the value parameter

An Example

```
import java.lang.reflect.*;
import java.awt.*;

class SampleSet {
    public static void main(String[] args) {
      Rectangle r = new Rectangle(100, 20);
      System.out.println("original: " +
                         r.toString());
      modifyWidth(r, new Integer(300));
      System.out.println("modified: " +
                         r.toString());
}
```

Example Continued

```
 static void modifyWidth(Rectangle r,
                         Integer widthParam ) {
  Field widthField;
  Integer widthValue;
  Class c = r.getClass();
  try {
       widthField = c.getField("width");
       widthField.set(r, widthParam);
      } catch (NoSuchFieldException e) {
               System.out.println(e);
      } catch (IllegalAccessException e) {
               System.out.println(e);
      }
  }
}
```

Obtaining Method Information

- To find out what public methods belong to a class, invoke the method named `getMethods`
- You can use a `Method` object to uncover a method's name, return type, parameter types, set of modifiers, and set of throwable exceptions
- With `Method.invoke`, you can even call the method itself

Example

```
import java.lang.reflect.*;
public class DumpMethods {
   public static void main(String args[]) {
   try {
        Class c = Class.forName(args[0]);
        Method m[] = c.getDeclaredMethods();
        for (int i = 0; i < m.length; i++)
             System.out.println(m[i].toString());
       } catch (Throwable e) {
             System.err.println(e); }
   }
}
```

Example Continued (invocation)

```
java DumpMethods java.util.Stack
```

```
public java.lang.Object
  java.util.Stack.push(java.lang.Object)
public synchronized java.lang.Object
  java.util.Stack.pop()
public synchronized java.lang.Object
  java.util.Stack.peek()
public boolean java.util.Stack.empty()
public synchronized int
  java.util.Stack.search(java.lang.Object)
```

Invoking Methods

- Create a `Method` object by invoking `getMethod` on the `Class` object
  - The `getMethod` method has two arguments: a `String` containing the method name, and an `array` of `Class` objects
- Invoke the method by calling invoke
  - The invoke method has two arguments: an `array` of argument values to be passed to the invoked method, and an object whose class declares or inherits the method

Example Start

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

class SampleInvoke {
  public static void main(String[] args) {
   String firstWord = "Hello ";
   String secondWord = "everybody.";
   String bothWords = append(firstWord, secondWord);
   System.out.println(bothWords);
  }
  public static String append(String firstWord,
                              String secondWord) {
   String result = null;
   Class c = String.class;
   Class[ ] parameterTypes = new Class[] {String.class};
   Method concatMethod;
   Object[ ] arguments = new Object[] {secondWord};
```

Continuing

```
 try {
   concatMethod = c.getMethod("concat",
                                parameterTypes);
   result = (String) concatMethod.invoke(firstWord,
                                      arguments);
 } catch (NoSuchMethodException e) {
          System.out.println(e);
 } catch (IllegalAccessException e) {
          System.out.println(e);
 } catch (InvocationTargetException e) {
          System.out.println(e); }
 return result;
}
```

Discovering Class Constructors

- To create an instance of a class, you invoke a special method called a constructor
  - Like methods, constructors can be overloaded and are distinguished from one another by their signatures
- You can get information about a class's constructors by invoking the `getConstructors` method, which returns an `array` of `Constructor` objects
- You can use the methods provided by the `Constructor` class to determine the constructor's name, set of modifiers, parameter types, and set of throwable exceptions
- You can also create a new instance of the `Constructor` object's class with the `Constructor.newInstance` method

Introduction to an Example

- The sample program that follows prints out the parameter types for each constructor in the `Rectangle` class. The program performs the following steps
  - It retrieves an `array` of `Constructor` objects from the `Class` object by calling `getConstructors`
  - For every element in the `Constructor array`, it creates an `array` of `Class` objects by invoking `getParameterTypes`
    - The `Class` objects in the array represent the parameters of the constructor
  - The program calls `getName` to fetch the class name for every parameter in the `Class array` created in the preceding step

Example Start

```
import java.lang.reflect.*;
import java.awt.*;

class SampleConstructor {
   public static void main(String[] args) {
    Rectangle r = new Rectangle();
    showConstructors(r);
  }
  static void showConstructors(Object o) {
    Class c = o.getClass();
    Constructor[] theConstructors= c.getConstructors();
    for (int i = 0; i < theConstructors.length; i++) {
         System.out.print("( ");
    Class[] parameterTypes =
                theConstructors[i].getParameterTypes();
```

Balance of Example

```
    for (int k = 0; k < parameterTypes.length; k ++) {
         String parameterString =
                              parameterTypes[k].getName();
         System.out.print(parameterString + " ");
   }
   System.out.println(")");
   }
 }
}
```

Output from Example

What's this?

```
( )
( int int )
( int int int int )
( java.awt.Dimension )
( java.awt.Point )
( java.awt.Point java.awt.Dimension )
( java.awt.Rectangle )
```

Creating Objects

- We have already seen in a previous slide how to create an object with `newInstance` - that creates an instance (implicitly with the non-arg constructor)
- There is also a way to invoke a constructor with arguments using a `Constructor` instance
- Involves several steps:
  - Create a `Constructor` object by invoking `getConstructor` on the `Class` object returning an `array` of `Constructors`
  - Create the object by invoking `newInstance` on the `Constructor` object
    - The `newInstance` method has one parameter: an `Object` array whose elements are the argument values being passed to the constructor

Example

```
import java.lang.reflect.*;
import java.awt.*;

class SampleInstance {
  public static void main(String[] args) {
   Rectangle rectangle;
   Class rectangleDefinition;
   Class[] intArgsClass = new Class[] {int.class,
                                       int.class};
   Integer height = new Integer(12);
   Integer width = new Integer(34);
   Object[] intArgs = new Object[] {height, width};
   Constructor intArgsConstructor;
```

More of the Example

```
 try {
   rectangleDefinition =
                    Class.forName("java.awt.Rectangle");
   intArgsConstructor =
      rectangleDefinition.getConstructor(intArgsClass);
   rectangle = (Rectangle)
              createObject(intArgsConstructor, intArgs);
  } catch (ClassNotFoundException e) {
    System.out.println(e);
  } catch (NoSuchMethodException e) {
    System.out.println(e);
  }
}
```

More of the Example

```
public static Object createObject(Constructor
                                        constructor,
                                  Object[] arguments)
{
    System.out.println ("Constructor: " +
                        constructor.toString());
    Object object = null;
    try {
      object = constructor.newInstance(arguments);
      System.out.println ("Object: " +
                          object.toString());
      return object;
     }
```

Example

```
    catch (InstantiationException e) {
        System.out.println(e);
    } catch (IllegalAccessException e) {
        System.out.println(e);
    } catch (IllegalArgumentException e) {
        System.out.println(e);
    } catch (InvocationTargetException e) {
        System.out.println(e);
    }
    return object;
  }
}
```

Dealing with Arrays

- Identifying Arrays
- Retrieving Component Types
- Creating Arrays
- Getting and Setting Element Values

Identifying Arrays

- If you aren't certain that a particular object is an array, you can check it with the `Class.isArray` method

```
static void printArrayNames(Object target) {
  Class targetClass = target.getClass();
  Field[] publicFields = targetClass.getFields();
  for (int i = 0; i < publicFields.length; i++) {
    String fieldName = publicFields[i].getName();
    Class typeClass = publicFields[i].getType();
    String fieldType = typeClass.getName();
    if (typeClass.isArray()) {
      System.out.println("Name: " + fieldName +
                         ", Type: " + fieldType);
    }
  }
}
```

Retrieving Component Types

- By invoking the `getComponentType` method against the Class object that represents an array, you can retrieve the component type of the array's elements:

```
static void printComponentType(Object array) {
  Class arrayClass = array.getClass();
  String arrayName = arrayClass.getName();
  Class componentClass = arrayClass.getComponentType();
  String componentName = componentClass.getName();
  System.out.println("Array: " + arrayName +
                     ", Component: " + componentName);
}
```

Creating Arrays

- Use `newInstance` on an array

```
static Object doubleArray(Object source) {
  int sourceLength = Array.getLength(source);
  Class arrayClass = source.getClass();
  Class componentClass = arrayClass.getComponentType();
  Object result = Array.newInstance(componentClass,
                                    sourceLength * 2);
  System.arraycopy(source, 0, result, 0, sourceLength);
  return result;
}
```

Getting and Setting Element Values

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

class SampleGetArray {
  public static void main(String[] args) {
    int[ ] sourceInts = {12, 78};
    int[ ] destInts = new int[2];
    copyArray(sourceInts, destInts);
    String[ ] sourceStrgs = {"Hello ", "there ",
                             "everybody"};
    String[ ] destStrgs = new String[3];
    copyArray(sourceStrgs, destStrgs);
  }
```

Example Continued

```
public static void copyArray(Object source,
                             Object dest) {
  for (int i = 0; i < Array.getLength(source); i++) {
      Array.set(dest, i, Array.get(source, i));
      System.out.println(Array.get(dest, i));
    }
}
```

Output 

```
12
78
Hello
there
everybody
```

Simulating the `instanceof` Operator

- In this next example, a Class object for class A is created
- Then class instance objects are checked to see whether they are instances of A
- We'll check two
  - `Integer(37)` is not
  - But `new A()` is

Example

```
class A {}

public class instance1 {
   public static void main(String args[]) {
   try {
        Class cls = Class.forName("A");
        boolean b1 = cls.isInstance(new Integer(37));
        System.out.println(b1);
        boolean b2 = cls.isInstance(new A());
        System.out.println(b2);
  } catch (Throwable e) {
    System.err.println(e); }
 }
}
```

Capabilities Not Available Using Reflection

- What are a class' subclasses?
  - Not possible due to dynamic class loading
- method is currently executing
  - Not the purpose of reflection
  - Other APIs provide this capability

Dynamic Facilities In Java

- Java is a more dynamic language than C or C++
  - It was designed to adapt to an evolving environment
- For example, one major problem with C++ in a production environment is a side-effect of the way that code is implemented
  - If company A produces a class library (a library of plug and play components) and company B buys it and uses it in their product, then if A changes its library and distributes a new release, B will almost certainly have to recompile and redistribute their own software
- In an environment where the end user gets A and B's software independently (say A is an OS vendor and B is an application vendor) problems can result
- For example, if A distributes an upgrade to its libraries, then all of the software from B will break. It is possible to avoid this problem in C++, but it is extraordinarily difficult and it effectively means not using any of the language's OO features directly

Considerations

- By making these interconnections between modules later, Java completely avoids these interconnection problems
- Java makes the use of the object-oriented paradigm much more straightforward
- Libraries can freely add new methods and instance variables without any effect on their clients

Four Myths of Reflection

- “Reflection is only useful for JavaBeans™ technology-based components”
- “Reflection is too complex for use in general purpose applications”
- “Reflection reduces performance of applications”
- “Reflection cannot be used with the 100% Pure Java™ certification standard”

"Reflection Is Only Useful for JavaBeans™ Technology-based Components"

- False
- Reflection is a common technique used in other pure object oriented languages like Smalltalk and Eiffel
- Benefits
  - Reflection helps keep software robust
  - Can help applications become more
    - Flexible
    - Extensible
    - Pluggable

"Reflection Is Too Complex for Use in General Applications"

- False
- For most purposes, use of reflection requires mastery of only several method invocations
- The skills required are easily mastered
- Reflection can significantly…
  - Reduce the footprint of an application
  - Improve reusability

"Reflection Reduces the Performance of Applications"

- False
- Reflection can actually increase the performance of code
- Benefits
  - Can reduce and remove expensive conditional code
  - Can simplify source code and design
  - Can greatly expand the capabilities of the application

"Reflection Cannot Be Used With the 100% Pure Java™ Certification Standard"

- False
- There are some restrictions
  - "The program must limit invocations to classes that are part of the program or part of the JRE"[3]

3. Sun Microsystems, 100% Pure Java™ Certification Guide, version 3.1, May 2000.

Review

- The JRE allows 4 ways to reference a class
  - The class' class definition
  - Class literals
  - The ***`instanceof`*** keyword
  - Reflection
- Reflection is the only pure runtime way
  - Provides full access to the object's capabilities
  - Provides runtime capabilities not otherwise available
  - Improves the quality of an application

Review

- Solves several design issues
  - Simplifies the static complexity of methods by providing elimination of…
    - Nested if/else constructs
    - The switch/case construct
- Reflection provides…
  - Reduced level of maintenance
  - Programs become…
    - Flexible
    - Extensible
    - Pluggable