Technical University of Cluj-Napoca
Computer Science Department

Object Oriented
Programming Techniques

Reflection Techniques

Spring 2017

Objective

- Reflection
  - Ability of a running program to examine
    (i) itself,
    (ii) its software environment and
    (iii) change its actions depending on what it finds
- Reflection gives dynamic access/inspection to internal information for classes loaded into the JVM
- Increases application flexibility
- Reflection - powerful tool
  - Allows building flexible code that can be assembled at run time without requiring source code links between components.
- Some aspects of reflection can be problematic
  - These aspects will be presented in this lecture

Metadata

- Metadata is data about data
- For reflection, the program needs to have a representation of itself – it needs access to **metadata**
- In OO systems, metadata is organized into objects called meta-objects
- **Introspection** - runtime examination of meta-objects

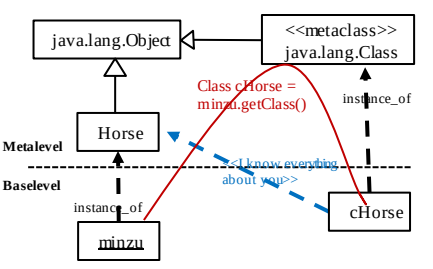
Metadata

- In Java, metadata about classes is stored in other classes
- Metadata for a class is stored in **java.lang.Class**
  - This is the entry point into reflection operations
- Metadata includes information about
  - The class itself, such as the package and superclass of the class
  - The interfaces implemented by the class.
  - Details of the constructors, fields, and methods defined by the class.

Class objects

- If we get an object of class Class, we may operate on the methods, constructors, fields of the type of that object
- How to get a Class object
  1. Use the method **getClass()** of the class Object if you have a reference to an object
  2. Use the static method **forName()** of the class Class (when the type is available, but no such object is available). This loads and initializes the specified class.

Metalevel and base level




```
Horse minzu = new Horse();
Class cHorse = minzu.getClass();
```

Metadata

Metadata is stored in classes

- Metadata for a class: java.lang.Class
- Metadata for a constructor: java.lang.reflect.Constructor
- Metadata for a field: java.lang.reflect.Field
- Metadata for a method: java.lang.reflect.Method

Metadata
Inter-dependencies

Inter-dependencies among reflection classes

```
class Class {
  Constructor[] getConstructors();
  Field getDeclaredField(String name);
  Field[] getDeclaredFields();
  Method[] getDeclaredMethods();
  …
}

class Field {
  Class getType();
  …
}

class Method {
  Class[] getParameterTypes();
  Class getReturnType();
  …
}
```

Class Class

Building Class objects using Object.getClass()

- getClass() method defined in class Object
- **final** method
  - What would happen if not final?
  - It can be overridden by programmer

Examples

```
Class c = "Alpha".getClass();
```

- Returns a Class object corresponding to the class String
- If p is a Point object

```
Point p = new Point (2.1, 3.2);
Class cp = p.getClass();
```

- Arrays are Objects
  - getClass() can be also invoked on array objects
- Another example

```
import java.util.HashSet;
import java.util.Set;
Set<String> s = new HashSet<>();
Class c = s.getClass();
```

- c is an object describing the java.util.HashSet

Class Class

forName

```
package foo;
public class Test {
   public Test () {
      System.out.println("Hello Test");
   }

   public static void main(String[] args) throws
Exception {
      Class cls = Class.forName("foo.Test");
      Test tst = (Test) cls.newInstance();
   }
}
```

Useful methods defined in class Class

```
class Class {

  public String getName(); // fully-qualified name

// Returns true if and only if the target Class object represents an array
public boolean isArray();

// Returns true if and only if the target Class object represents an interface
  public boolean isInterface();

// Returns true if and only if the target Class object represents a primitive type or void
  public boolean isPrimitive();

// If the target object is a Class object for an array, returns the Class object representing the component type
  public Class getComponentType(); // only for arrays
…
}
```


Metadata for primitive types and arrays

- Java associates a Class instance with each primitive type:

```
Class c1 = int.class; // c1 is Class object for primitive type int
Class c2 = boolean.class;
Class c3 = void.class;
Class c4 = byte.class;
```

- Syntactically, any class name followed by .class evaluates to a class object

```
Class c5 = Integer.class
```

Use Class.forName() to access the Class object for an array

```
Class c5 = Class.forName("[B"); // byte[]
Class c6 = Class.forName("[[B"); // byte[][]
```

Encoding scheme used by Class.forName()

- B -> byte; C -> char; D -> double; F -> float; I -> int;
- J -> long;
- Lclass-name -> class-name[];
- S -> short;
- Z -> boolean
- Use as many "["s as there are dimensions in the array

Introspecting about mthods

Methods defined in class Class for introspecting about methods

Method getMethod (String name, Class[] parameterTypes)

- Returns a Method object that represents a public method (either declared or inherited) of the target Class object having the name specified by the first parameter and the signature specified by the second parameters

Method[] getMethods()

- Returns an array of Method objects that represent all of the **public** methods (either declared or inherited) supported by the target Class object

Method getDeclaredMethod (String name, Class[] parameterTypes)
Method[] getDeclaredMethods()

- These two methods are similar with the above ones but return only methods defined in the class (no inheritance) and having **all** qualifiers (public, private, etc.)

Exception thrown: *NoSuchMethodException*

Introspecting about mthods

- **Usage**

-Retrieve the method:
ClassObject.**getMethod**(String name, Class[] parameterTypes)
-Call the method:
MethodObject.**invoke**(Object target, Object[] parameters)
(will be back on method invocation)

Example

```
// get the Class object
Object obj = …
Class cls = obj.getClass();

// Get the method
Method m = cls.getMethod(“ doWork”,
                  new Class[]{ String.class, String.class});
```

- The second parameter represents the array of Class objects of the method parameters (one array entry for each parameter)

```
// Call the method
Object result= m.invoke(obj, new Object[]{“ x”,“y”} );
```

Methods retrieval (querying methods)
Representing primitive types parameters

Example class Vector

```
public class Vector ... {
  public synchronized boolean addAll (Collection c) ...
  public synchronized void copyInto (Object[] anArray) ...
  public synchronized Object get (int index) ...
}
```

Exampe of querrying class Vector for its method **get:**

```
Method m = Vector.class.getMethod("get", new Class[] {int.class});
```

-A class object that represents a primitive type can be identified using *isPrimitive*.

Methods retrieval (querying methods)
Representing interfaces parameters

Querying the Vector class for its addAll method:

```
Method m = Vector.class.getMethod("addAll", new Class[]
{Collection.class});
```

- The *isInterface* method of Class can be used to identify class objects that represent interfaces.

Methods retrieval (querying methods)
Representing array types parameters

Querying the Vector class for its copyInto method:
Method m = Vector.class.getMethod("copyInto", new Class[]{Object[].class});

- The *isArray* method of Class can be used to identify class objects that represent arrays
- The component type for an array class can be obtained using *getComponentType*
- Java treats multidimensional arrays like nested single-dimension arrays.
- Therefore, the line
- int[][].class.getComponentType()

evaluates to int[].class

- Note the distinction between component type and element type. For the array type int[][], the component type is int[] while the element type is int.

Working with method objects

The class of the metaobjects that represents methods is **java.lang.reflect.Method**

Main methods:

Method	Description
Class getDeclaringClass()	Returns the Class object that declared the method represented by this Method object
Class[] getExceptionTypes()	
int getModifiers()	Returns the modifiers for the method represented by this Method object encoded as an int
String getName()	
Class[] getParameterTypes()	Returns an array of Class objects representing the formal parameters in the order in which they were declared
Class getReturnType()	
Object invoke(Object obj, Object[] args)	Invokes the method represented by this Method object on the specified object with the arguments specified in the Object array

Invoking methods dynamically

- Dynamic invocation - enables a program to call a method on an object at runtime, based on runtime introspection, without specifying which method at compile time

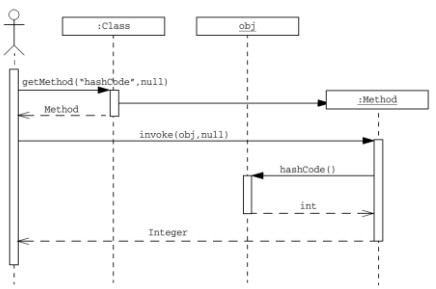
```
public static void setObjectProperty (Object o, Property p) {
 Class cls = o.getClass();
 try {
  Method m = cls.getMethod("setProperty",
                           new Class[] {Property.class});
  m.invoke (o, new Object[] { p } );
 }
  catch { … }
}
```

- In the above invocation:
  - p is a variable of type Property (eg. mass, height, length, density, diameter, etc.
  - Object[] is an array of arguments that are passed as actual parameters in method m invocation
- If method has no parameters – **null** can be used or a zero length array
- If *setProperty* is static method of class of **o,** the first parameter is ignored (no target objects for static metods)
  - In this case null can be first parameter
- **Important**. The return value of invocation is Object

Invoking methods dynamically
Using primitives

- Primitive types used as parameters are wrapped before calling (e.g. int is wrapped to Integer)
- Return type is wrapped before effective returning
- Example:

```
Method method = obj.getClass().getMethod("hashCode", null);
int code = ((Integer) method.invoke(obj, null)).intValue();
```



Example

- Set an object property without knowing its concrete type
- It's a non-intrusive method (not invading the code of any existing method)
- Example – setObjectProperty
- Property can be Length, Density, Color, etc.

```
public static void setObjectPoperty (Object o, Property p) {
  Class cls = o.getClass(); // 1
  try {
    Method m = cls.getMethod("setProperty",
                                    new Class[] { Property.class} ); //2
    m.invoke (o, new Object[] { p } ); // 3
  }
   catch (NoSuchMethodException ex) {  // 4
      throw new IllegalArgumentException (cls.getName() +
      " does not support method setProperty(Property)" );
   }
  catch (IllegalAccessException ex) { // 5
      throw new IllegalArgumentException (
       "Insufficient access permissions to call " +
      " setProperty (: Property) in class " + cls.getName());
   }
  catch(InvocatonTargetException ex) {  // 6
      throw new RuntimeException(ex);
   }
}
```


Invoking methods dynamically
Exceptions

- IllegalAccessException - The class calling **invoke** has no appropriate access privileges for the method
  - For example, when attempting to invoke a private method from outside its declaring class
- IllegalArgumentException can be thrown by invoke under several circumstances.
  - Supplying an invocation target whose class does not support the method being invoked.
  - Supplying an args array of incorrect length or with entries of the wrong type
- InvocationTargetException - If any exception is thrown by the method being invoked, that exception is wrapped in an and then thrown

Introspecting class hierarchies

- **Class** methods for dealing with inheritance
- The **getInterfaces()** method returns Class objects that represent interfaces
  - When called on a Class object that represents a **class**, *getInterfaces* returns Class objects for interfaces specified in the **implements** clause of that class's declaration
  - When called on a Class object that represents an **interface**, *getInterfaces* returns Class objects specified in the **extends** clause of that interface's declaration.

Method	Description
Class[] getInterfaces()	Returns an array of Class objects that represent the direct superinterfaces of the target Class object
Class getSuperclass()	Returns the Class object representing the direct superclass of the target Class object or null if the target represents Object, an interface, a primitive type, or void
boolean isAssignableFrom (Class cls)	Returns true if and only if the class or interface represented by the target Class object is either the same as or a superclass of or a superinterface of the specified Class parameter
boolean isInstance (Object obj)	Returns true if and only if the specified Object is assignment-compatible with the object represented by the target Class object

Reflectively constructing objects

- **Class** methods for constructor introspection
- Example

cls.getConstructor(new Class[] {String.class, String.class})

- Query for a public constructor that takes two String objects as parameters
- NoSuchMethodException is thrown if there is no constructor declared for the parameter list specified

Note. Class objects returned by **forName** may be used in the specification of a parameter list

Method	Description
Constructor getConstructor (Class[ ] parameterTypes)	Returns the **public** constructor with specified argument types if one is supported by the target class
Constructor getDeclaredConstructor (Class[ ] parameterTypes)	Returns the constructor with specified argument types if one is supported by the target class
Constructor[ ] getConstructors()	Returns an array containing all of the **public** constructors supported by the target class
Constructor[ ] getDeclaredConstructors()	Returns an array containing all of the constructors supported by the target class

Reflectively constructing objects

- **newInstance** method of class Class builds an object of the target class' object
- newInstance is similar with calling default constructor (or constructor with no arguments)
- Java Reflection API defines a metaclass for dealing with constructors **java.lang.Reflect.Constructor**

Constructor metadata – reflective methods

- **Constructor** is the class of metaobjects that represents Java constructors
- The interface to Constructor is very much like the interface to Method, except it supports a **newInstance** method instead of **invoke**
- The reflective methods of Constructor is shown below (note that *newInstance* works similar with the same method in Class)

Method	Description
Class getDeclaringClass()	Returns the class object that declares the constructor represented by this Constructor
Class[ ] getExceptionTypes()	Returns a Class array representing the types of exceptions that can be thrown from the body of this Constructor
int getModifiers()	Returns a bit vector encoding the modifiers present and absent for this member
String getName()	Returns the name of the constructor
Class[ ] getParameterTypes()	Returns a Class array representing the parameter types that are accepted by this constructor in order
Object newInstance (Object[ ] initargs)	Invokes the constructor with the specified parameters and returns the newly constructed instance

Reflectively accessing fields

- **Class** methods for fields introspection
- If parameters for either getField or getDeclaredField specify a field that does not exist, these methods throw **NoSuchFieldException**.
- Querying for fields can be disabled in the **Java security manager** If this feature is disabled, all of these methods throw a **SecurityException**
- The return type of the methods is **java.lang.reflect.Field** which is a metaobject giving information about field's name, declaring class, and modifiers. Field also provides several methods for getting and setting values.

Method	Description
Field getField (String name)	Returns a Field object that represents the specified **public** member field of the class or interface represented by this Class object
Field[] getFields()	Returns an array of Field objects that represents all the accessible **public** fields of the class or interface represented by this Class object
Field getDeclaredField (String name)	Returns a Field object that represents the specified declared field of the class or interface represented by this Class object
Field[] getDeclaredFields()	Returns an array of Field objects that represents each field declared by the class or interface represented by this Class object

Reflectively accessing fields

- Obtaining the values of all fields of an object (declared and **inherited**)
- Useful for example when serializing the object (obj)

```
public static Field[] getInstanceVariables1(Class cls) {
  List accum = new LinkedList();
 while (cls != null) {
   Field[] f = cls.getDeclaredFields();
   for (int i=0; i<f.length; i++) {
    accum.add(f[i]);
   }
   cls = cls.getSuperclass();
 }
 Field[] allFields = (Field[]) accum.toArray(new
                                    Field[accum.size()]);
 return allFields;
}
```

Reflectively accessing fields

- Methods defined by metacass **Field**

Method	Description
Class getType()	Returns the Class object that represents the declared type for the field represented by this Field object
Class getDeclaringClass()	Returns the Class object that declared the field represented by this Field object
String getName()	Returns the name of the field represented by this Field object
int getModifiers()	Returns the modifiers for the field represented by this Field object encoded as an int
Object get (Object obj)	Returns the value in the specified object of the field represented by this Field
boolean getBoolean (Object obj) ...	Returns the value in the specified object of the Boolean field represented by this Field
void set (Object obj, Object value)	Sets the field of the specified object represented by this Field object to the specified new value
void setBoolean (Object obj, boolean value) ...	Sets the field of the specified object represented by this Field object to the specified boolean value

Reflectively accessing fields

- If **field** refers to a field object, we can use the methods from the table to get the values necessary to identify it uniquely:

```
String fieldName = field.getName();
String fieldDeclClass = field.getDeclaringClass().getName();
```

- For example, this string information could be stored in the serialized form along with the value of the field
- The de-serializer may work in the opposite way
- At deserialization time, the class specified in fieldDeclClass may need to be loaded

Reflectively accessing fields
Getting and setting field values

- If **field** refers to a field object of the object **obj**, its value is accessed using:

```
Object value = field.get(obj);
```

- If the field type is primitive, Java wraps the value in an appropriate wrapper object
- Alternatively, knowing the type of primitive, the code can access the value directly using one of the primitive access methods (**getBoolean** and so on).
- The following line sets the value of the field back to the value just extracted

```
field.set(obj, value);
```

- If the type of field is primitive, wrapping the value in the appropriate wrapper class allows successful use of the set method
- There is also a corresponding group of methods for each primitive, which do not require wrapping
- The **set** method is also useful (for deserialization for example)

Reflectively accessing fields
Exceptions

- IllegalArgumentException
  - If the field is not defined or inherited by the object in the first argument to get/set
  - If a set method is called with a value argument that is not assignable to the field
  - If a primitive get method is called and the value of the field cannot be converted into that primitive
- IllegalAccessException
  - If the class calling get or set does not have visibility into the field.
- Note. These access checks can be suppressed

Reflectively examining modifiers

- In the previous code **getInstanceVariables1**, allFields array accumulates all fields no matter their modifiers
- Some applications need to know the type of the modifiers
- For example, in object serialization we are not interested in static variables but instance variables (which are non-static).

Interface **Member** methods

- *getModifiers* – return the modifiers as int where the following 11 modifiers are represented: public, static, native, volatile, protected, abstract, synchronized, strictfp, private, final, transient
- Class **Modifiers** – defines methods (return type Boolean) for identify the modifiers:
  static boolean isPublic(int mod) - returns true if and only if the public modifier is present in the set of modifiers represented by the int argument
- Other methods: isPrivate, isProtected, isStatic, isFinal, isAbstract, etc.

Method	Description
Class getDeclaringClass()	Returns the Class object that declared the member
String getName()	Returns the name of the member
int getModifiers()	**Returns the modifiers for the member encoded as an int**

Reflectively examining modifiers

- The **getInstanceVariables** method below traverses up the inheritance hierarchy, accumulating declared fields on the way up
- Filtering static fields: use *getModifiers* and *Modifier.isStatic*
- The returned array has the set of non-static fields for the class.

```
public static Field[] getInstanceVariables(Class cls) {
 List accum = new LinkedList();
 while (cls != null) {
  Field[] fields = cls.getDeclaredFields();
  for (int i=0; i<fields.length; i++) {
   if (!Modifier.isStatic(fields[i].getModifiers())) {
    accum.add(fields[i]);
   }
  }
  cls = cls.getSuperclass();
 }
 Field[] retvalue = new Field[accum.size()];
 return (Field[]) accum.toArray(retvalue);
}
```

Accessing non-public members

- Access checks can be enabled or disabled (suppressed)
- If enabled, Java reflexive methods cannot access private members
- The class **java.lang.reflect.AccessibleObject** is the parent class of both **Field** and **Method** metaclasses
- **AccessibleObject** defines method **setAccessible** that enables/disables runtime access checking
- For **field** of type Field, the following, disables all JVM runtime access checks on uses of the metaobject referred to by field.

field.setAccessible(true);

- This allows reflective access to its value from outside the scope of its visibility
- Seting parameter **false** re-enables the runtime access checks.
- Getting access to **field**:

```
if (!Modifier.isPublic(field.getModifiers())) {
 field.setAccessible(true);
}
Object value = field.get();
```


Accessing non-public members

- **AccessibleObject** methods

Method	Description
void setAccessible (boolean flag)	Sets the accessible flag of the target object to the value of the argument
boolean isAccessible()	Returns true if and only if the value of the accessible flag of the target object is true
static void setAccessible (AccessibleObject[] array, boolean flag)	Sets the accessible flags for each element of an array of accessible objects

Accessing non-public members

Final Notes

- Setting objects as accessible can be disabled in the security manager
- If this feature has been disabled, the setAccessible methods each throw a SecurityException.
- The default security manager permits the use of setAccessible on members of classes loaded by the same class loader as the caller. Supplying a custom security manager can change this policy
- For details on security managers, check Java documentation

Reflecting on arrays

- Java provides **java.lang.reflect.Array** for performing reflective operations on all array objects (arrays of objects and arrays of primitives)
- Assume obj refers to an array
- The length of an array

```
int length = Array.getLength(obj);
```

- Reflective access on the ith element of the array (if the component type of the array is primitive, **get** wraps the accessed value in its corresponding wrapper).

```
Array.get(obj, i)
```

- Other methods defined by class Array

Reflecting on arrays

- Main **java.lang.reflect.Array** methods

Method	Description
Object newInstance (Class componentType, int length)	Creates a new array that has the specified component type and length.
Object newInstance (Class elementType, int[] dimensions)	Creates a new array that has the specified element type and dimensions.length dimensions.
int getLength(Object array)	Returns the number of components of the specified array.
boolean getBoolean (Object array, int index) ...	If the component type of the specified array is boolean, the component value at index is returned.
void set(Object array, int index, Object value)	Sets the component at index to the specified value. Unwraps primitives, if necessary.
void setBoolean (Object array, int index, boolean value) ...	If the component type of the specified array is boolean, the component at index is set to the specified value.
Object get (Object array, int index)	Returns the component value at index. Wraps primitives if necessary.

Where reflection is used

Auto-completion in a text editor

- Some Java editors and IDEs provide auto-completion
- Example: you type "someObj." and a pop-up menu lists fields and methods for the object's type
- The pop-up menu is populated by using Java reflection

Junit

- JUnit 3 uses reflection to find methods whose names start with "test"
- The algorithm was changed in JUnit 4
- Test methods are identified by an annotation (annotations were introduced in Java 1.5)
- Reflection is used to find methods with the appropriate annotation
- Naming conventions of methods are used to infer semantics (JUnit test methods)

Ant

- Ant reads build (compilation) instructions from an XML file

Where reflection is used

Spring

- Spring is open source framework for developing Java applications
- Spring uses reflection to create an object for each bean
  - The object's type is specified by the class attribute
- By default, the object is created with its default constructor
  - You can use constructor-arg elements (nested inside bean) to use a non-default constructor
- After an object is constructed, each property is examined
  - Spring uses reflection to invoke obj.setXxx(value) - where Xxx is the capitalized name of property xxx
  - Spring uses reflection to determine the type of the parameter passed to obj.setXxx()
  - Spring can support primitive types and common Collection types
  - The ref attribute refers to another bean identified by its id

Where reflection is used

Code generators

- Most compilers have the following pipeline architecture: input file -> parser -> parse tree -> code-generation -> output the generated code to files
- Java's reflection metadata is conceptually similar to a parse tree
- You can build a **Java code generation tool** as follows:
  - ***Do not* write a Java parser**. Instead run the Java compiler
  - Treat generated **.class** files as your parse tree
  - Use reflection to navigate over this "parse tree"

Where reflection is used

Java bytecode manipulation

- **Optimization:**
  - Read a .class file, optimize bytecode and rewrite the .class file
- **Code analysis:**
  - Read a .class file, analyze bytecode and generate a report
- **Code obfuscation:**
  - Mangle names of methods and fields in .class files
- **Aspect-oriented programming (AOP):**
  - Modify bytecode to insert "interception" code
  - Generate proxies for classes or interfaces
  - Spring uses this technique

XML serializing objects

- JVM built-in serializer
  - Classes need to implement the interface Serializable => problems with objects from classes defined in 3 rd party libraries that are not implemented Serializable
  - Generates binary format
  - Can be used only with Java platform => Cannot be used to send objects to other platforms
- Advantages of custom serializer
  - No need to implement interface Serializable
  - May generate text that can be read by humans - corresponding text can be used in debugging
  - Can be used to send objects to other platforms

XML serializing objects

- XML
  - Self-describing text format for encoding structured nested data
  - All previously mentioned advantages
  - Large industry support for parsing, presentation, and Web services
- XML structure
  - XML element – delimited by a pair of tags (opening and closing)
- Example

```
<menu>
<food>
   <name>Letuce Soup</name>
   <price>6</price>
   <description> Good and healthy soup </description>
   <calories>200</calories>
</food>
<food>
   <name>Beef Steak</name>
   <price>18.50</price>
   <description>Good well done steak</description>
   <calories>1500</calories>
</food>
<food>
   <name>Mineral water</name>
   <price>2</price>
   <description>Sparkling water</description>
   <calories>50</calories>
</food>
</menu>
```


XML serializing objects

- An element with no other elements or text `<menu />`
- The opening tag of an element may contain name-value pairs called **attributes**
- Example of an empty element with an attribute

```
<tag-name attribute-name="attribute value" />
```

- Each file, string, or stream of well-formed XML is called a **document**
- An XML document has one element called its **root element** under which all other document content falls.
- There are Java libraries dealing with XML documents
- Our examples uses JDOM (see www.jdom.org).

XML serializing objects

A user defined method for object serialization must:

1. Get from its argument (the object to be serialized) a list of its fields.
2. Find the declaring class and field name that uniquely identifies each field
3. Get the value for each field.
4. If the field value is an object and it has not already been serialized, serialize that object.
5. If the field value is a primitive, store the value in a way that it can easily be retrieved

Exercise

Implement a custom method for object serialization:

public static Document serializeObject (Object source) throws Exception { … }

Example 1
(from formally sun.java.com)

```
import java.lang.annotation.Annotation;
import java.lang.reflect.Modifier;
import java.lang.reflect.Type;
import java.lang.reflect.TypeVariable;
import java.util.Arrays;
import java.util.ArrayList;
import java.util.List;
import static java.lang.System.out;
public class ClassDeclarationSpy {
   public static void main(String... args) {
     try {
        Class<?> c = Class.forName(args[0]);
        out.format("Class:%n  %s%n%n", c.getCanonicalName());
        out.format("Modifiers:%n  %s%n%n", Modifier.toString(c.getModifiers()));
        out.format("Type Parameters:%n");
        TypeVariable[] tv = c.getTypeParameters();
        if (tv.length != 0) {
             out.format("  ");
             for (TypeVariable t : tv)
                out.format("%s ", t.getName());
             out.format("%n%n");
        } else {
             out.format("  -- No Type Parameters --%n%n");
        }
```

Example 1 (cont.)

```
        out.format("Implemented Interfaces:%n");
        Type[] intfs = c.getGenericInterfaces();
        if (intfs.length != 0) {
            for (Type intf : intfs)
                out.format("  %s%n", intf.toString());
            out.format("%n");
        } else {
            out.format("  -- No Implemented Interfaces --%n%n");
        }

        out.format("Inheritance Path:%n");
        List<Class> l = new ArrayList<Class>();
        printAncestor(c, l);
        if (l.size() != 0) {
            for (Class<?> cl : l)
                out.format("  %s%n", cl.getCanonicalName());
            out.format("%n");
        } else {
            out.format("  -- No Super Classes --%n%n");
        }
```


Example 1 (cont.)

```
        out.format("Annotations:%n");
        Annotation[] ann = c.getAnnotations();
        if (ann.length != 0) {
            for (Annotation a : ann)
                out.format("  %s%n", a.toString());
            out.format("%n");
        } else {
            out.format("  -- No Annotations --%n%n");
        }
      // production code should handle this exception more gracefully
      } catch (ClassNotFoundException x) {
        x.printStackTrace();
      }
    }
private static void printAncestor(Class<?> c, List<Class> l) {
    Class<?> ancestor = c.getSuperclass();
    if (ancestor != null) {
      l.add(ancestor);
      printAncestor(ancestor, l);
    }
  }
}
```

Example 1 (cont.)
The output (1)

```
> java ClassDeclarationSpy java.util.concurrent.ConcurrentNavigableMap
Class:
  java.util.concurrent.ConcurrentNavigableMap

Modifiers:
  public abstract interface

Type Parameters:
  K V

Implemented Interfaces:
  java.util.concurrent.ConcurrentMap<K, V>
  java.util.NavigableMap<K, V>

Inheritance Path:
  – No Super Classes –

Annotations:
  – No Annotations –
The actual declaration for java.util.concurrent.ConcurrentNavigableMap is:
public interface ConcurrentNavigableMap<K,V>
    extends ConcurrentMap, NavigableMap<K,V>
```


Example 1 (cont.)
The output (2)

```
> java ClassDeclarationSpy java.io.InterruptedIOException
Class:
  java.io.InterruptedIOException

Modifiers:
  public

Type Parameters:
  – No Type Parameters –

Implemented Interfaces:
  – No Implemented Interfaces –

Inheritance Path:
  java.io.IOException
  java.lang.Exception
  java.lang.Throwable
  java.lang.Object

Annotations:
  – No Annotations –
```

Example 1 (cont.)
The output

```
$ java ClassDeclarationSpy java.security.Identity
Class:
 java.security.Identity

Modifiers:
 public abstract

Type Parameters:
 – No Type Parameters –

Implemented Interfaces:
 interface java.security.Principal
 interface java.io.Serializable

Inheritance Path:
 java.lang.Object

Annotations:
 @java.lang.Deprecated()
```


Example 2
(from sun.java.com)

```
import java.lang.reflect.Constructor;
import java.lang.reflect.Field;
import java.lang.reflect.Method;
import java.lang.reflect.Member;
import static java.lang.System.out;

enum ClassMember { CONSTRUCTOR, FIELD, METHOD, CLASS, ALL }

public class ClassSpy {
   public static void main(String... args) {
      try {
         Class<?> c = Class.forName(args[0]);
         out.format("Class:%n  %s%n%n", c.getCanonicalName());

         Package p = c.getPackage();
         out.format("Package:%n  %s%n%n",
                 (p != null ? p.getName() : "– No Package –"));
```

Example 2 (cont.)

```
        for (int i = 1; i < args.length; i++) {
            switch (ClassMember.valueOf(args[i])) {
            case CONSTRUCTOR:
                printMembers(c.getConstructors(), "Constructor"); break;
            case FIELD:
                printMembers(c.getFields(), "Fields"); break;
            case METHOD:
                printMembers(c.getMethods(), "Methods"); break;
            case CLASS:
                printClasses(c); break;
            case ALL:
                printMembers(c.getConstructors(), "Constuctors");
                printMembers(c.getFields(), "Fields");
                printMembers(c.getMethods(), "Methods");
                printClasses(c);
                break;
            default:
                assert false;
            }
        }
      // production code should handle these exceptions more gracefully
    } catch (ClassNotFoundException x) {
        x.printStackTrace();
    }
  }
```


Example 2 (cont.)

```
private static void printMembers(Member[] mbrs, String s) {
    out.format("%s:%n", s);
    for (Member mbr : mbrs) {
        if (mbr instanceof Field)
            out.format("  %s%n", ((Field)mbr).toGenericString());
        else if (mbr instanceof Constructor)
            out.format("  %s%n", ((Constructor)mbr).toGenericString());
        else if (mbr instanceof Method)
            out.format("  %s%n", ((Method)mbr).toGenericString());
    }
    if (mbrs.length == 0)
        out.format("  -- No %s --%n", s);
    out.format("%n");
  }
  private static void printClasses(Class<?> c) {
    out.format("Classes:%n");
    Class<?>[] clss = c.getClasses();
    for (Class<?> cls : clss)
        out.format("  %s%n", cls.getCanonicalName());
    if (clss.length == 0)
        out.format("  -- No member interfaces, classes, or enums --%n");
    out.format("%n");
  }
}
```

Example 2 (from sun.java.com) The output (1)

```
$ java ClassSpy java.lang.ClassCastException CONSTRUCTOR
Class:
  java.lang.ClassCastException

Package:
  java.lang

Constructor:
  public java.lang.ClassCastException()
  public java.lang.ClassCastException(java.lang.String)
```