326.041 (2015S) – Practical Software Technology

(Praktische Softwaretechnologie)

Reflection, Scripting, Serialization

Alexander Baumgartner
Alexander.Baumgartner@risc.jku.at

Research Institute for Symbolic Computation (RISC)
Johannes Kepler University, Linz, Austria

Reflection

- Code which is able to **inspect** other **code** (or itself).
- Reflection is a key strategy for **metaprogramming.**
  - Programs with the ability to treat other programs as their data.
  - Self-modifying code.
  - Adapt a given program to different situations dynamically.
- Reflection is often used as part of **software testing.**
- Reflection can be used to **override** member **accessibility** rules.
  - Change the value of a private field in a third-party library's class.
  - **Dangerous!** You need to know the internals of the library.

Drawbacks of Reflection

There are three major drawbacks of reflection:

- **Performance Overhead:** No optimization of the executed code.
- **Security Restrictions:** SecurityManager might forbid reflection.
- **Exposure of Internals:** Unexpected side-effects, which may render code dysfunctional.

Retrieving Class Objects

- **java.lang.Class** is the entry point for all the reflection operations.
- We already know how to retrieve the class of a certain type or object:
  - The class of a type Integer can be obtained by **Integer.class**.
  - The class of an array, e.g. int[][][], can be obtained by **int[][][].class**.
  - The class of a primitive type int can also be obtained by **int.class**.
  - The class of an object foo can be obtained by **foo.getClass()**.
- **Class.forName("..."):** It is possible to obtain a class from a String which contains a class name.
  - **Only** works for **reference types.**
  - Class.forName("`java.lang.Integer`") returns class for type Integer.
  - Class.forName("`[[[I`") returns class for type int[][][].
  - Class.forName("`[Ljava.lang.String`") returns class for type String[]
  - The name is the same as returned by the method getName().

`http://docs.oracle.com/javase/7/docs/api/java/lang/Class.html#getName()`

Instantiating an Object

- First we need a class object. E.g.:

```
1 Class<?> c = "".getClass(); // OR
2 Class<?> c = String.class;  // OR
3 Class<?> c = Class.forName("java.lang.String");
```

- We can create new instances by calling the method newInstance():

```
1 Class<?> c = Class.forName("java.lang.String");
2 Object   o = c.newInstance();
3 System.out.println(o);       // Empty String object
```

  - c.newInstance() invokes the default constructor String().

```
1 public String() { this.value = new char[0]; }
```

  - Given some class name. Is there a default constructor?
  - If a default constructor exists, is it public?

Obtaining a Constructor

- E.g.: Polynomial has the following constructor:

```
1 public Polynomial(int numVars) { this.numVars = numVars;}
```

- We **cannot** create new instances by calling the method newInstance():

```
1 c = Class.forName("at.jku.teaching.swtech.Polynomial");
2 o = c.newInstance(); // java.lang.InstantiationException
```

The information $numVars$ is missing.

```
1 c = Class.forName("java.lang.Math");
2 o = c.newInstance(); // java.lang.IllegalAccessException
```

The constructor Math() is private.

- **getConstructors()** returns an array of **public** constructors.
- getConstructor(Class...) returns the corresponding **public** constructor.
- **getDeclaredConstructors()** returns an array of all the constructors.
- getDeclaredConstructor(Class...) returns corresponding constructor.

Access Modifiers

- The method getModifiers() returns an int value which encodes all the access modifiers:

```
1 Class<?> c = Class.forName("java.lang.Math");
2 Constructor<?> cons = c.getDeclaredConstructor();
3 int classM = c.getModifiers();
4 int consM  = cons.getModifiers();
```

- Use the class java.lang.reflect.Modifier to query information:

```
1 System.out.println(Modifier.isPublic(classM));//true
2 System.out.println(Modifier.isFinal(classM)); //true
3 System.out.println(Modifier.isPublic(consM)); //false
4 System.out.println(Modifier.isPrivate(consM));//true
```

- Make a constructor, method, field accessible for reflection:

```
1 cons.setAccessible(true);
2 cons.newInstance();        //Invoke private constructor
```

Invoke Constructor with Arguments

- Back to the example of the polynomial.

```
1 public Polynomial(int numVars) { this.numVars = numVars;}
```

- Obtain a public constructor and invoke it with an argument.

```
1 Class<?> c = Class.forName("at.jku...Polynomial");
2 Constructor<?> cons = c.getConstructor(int.class);
3 Object o = cons.newInstance(2);
```

Invoke Method / Read and Write Field

- **Invoke Method:** Similar to invoking a constructor.
  - getMethods() returns an array of public methods.
  - getMethod(String, Class...) returns the public method with the corresponding **name and argument types**.
  - getDeclaredMethods() returns an array of all the methods.
  - getDeclaredMethod(String, Class...) returns corresponding method.

```
1 String s = "Hello";
2 Method m = s.getClass().getMethod(
3             "replace", char.class, char.class);
4 System.out.println(m.invoke(s, 'e', 'a'));
```

- **Fields:** Obtain Field by getFields(), getField(String), getDeclaredF...
- Getter methods to obtain a value and Setter methods to set a value:

```
1 char[] ca = {'B', 'y', 'e'};
2 Field m = s.getClass().getDeclaredField("value");
3 m.setAccessible(true);
4 System.out.println(((char[])m.get(s))[0]);
5 m.set(s, ca);
```

Example

- Create a Java class Form.java to represent a form which should be filled out by the user.
- Provide Getter and Setter methods for the fields.
- Read all the Getter to generate a form GUI.

```
1 for (Method method : methods)
2     if (method.getName().startsWith("get") &&
3             !method.getName().equals("getClass"))
4         System.out.println(method.getName().substring(3) + ":");
```

- Process the form response (e.g. HTTP response) by invoking the Setter which corresponds to the input field name.

If you add a new field to Form.java you don't need to update the GUI.

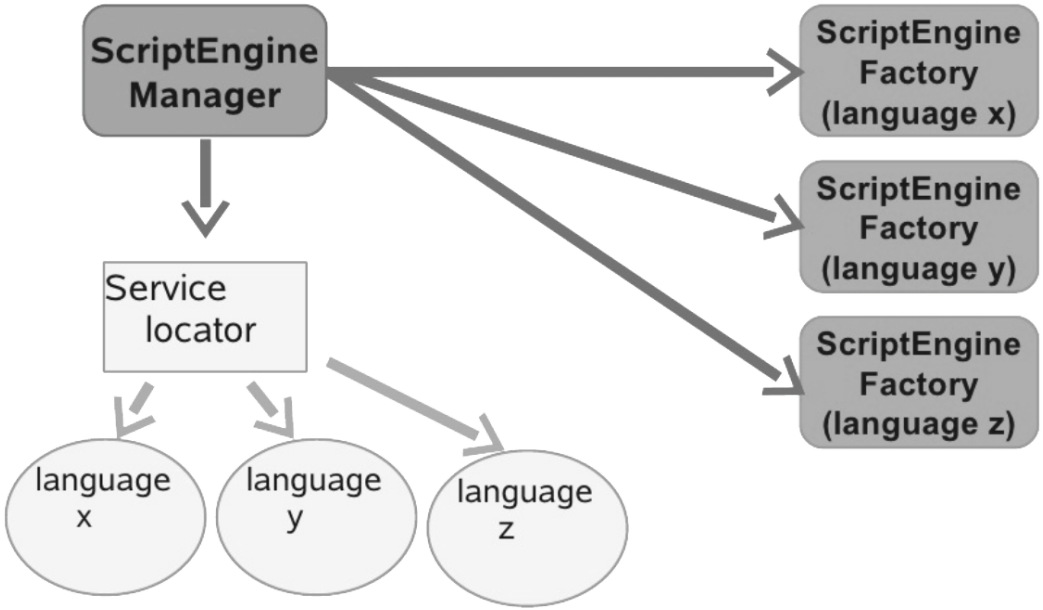
Scripting

- Java has build in scripting support. (Rhino script engine.)
  - It ships with JavaScript as reference implementation.
- Scripts have full access to the Java API.
  - Import and use Java classes.
  - Implement Java interfaces.
  - ...
- Load a script from a file and execute it in your Java program. Example:

```
1 ScriptEngineManager em = new ScriptEngineManager();
2 ScriptEngine engine = em.getEngineByName("JavaScript");
3 engine.eval("print('Hello')");
4 engine.eval(new FileReader(fileName));
```

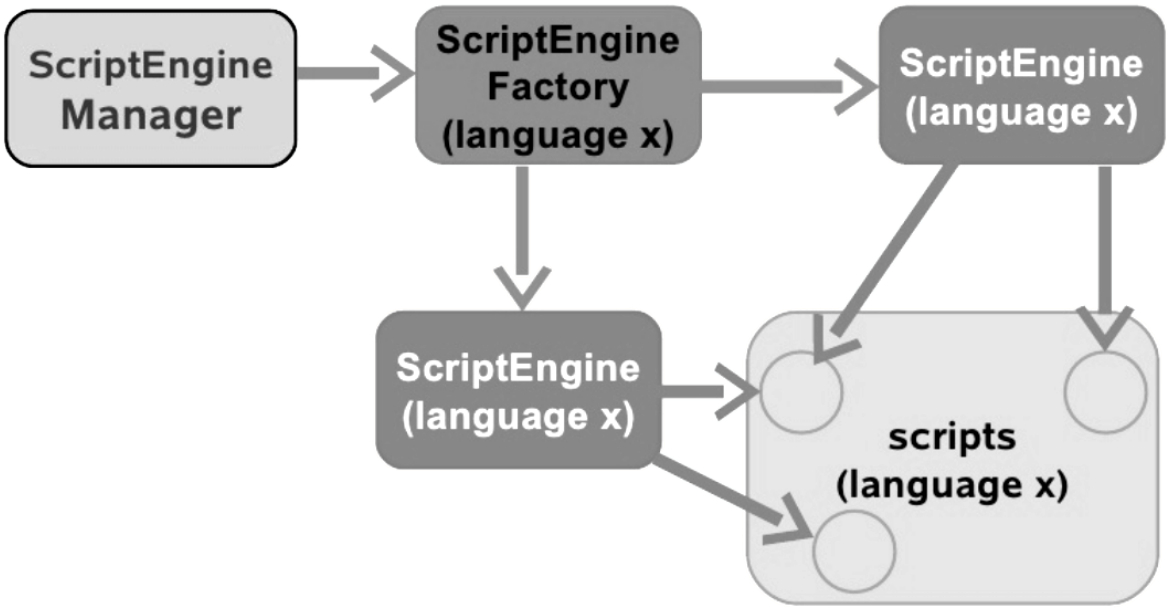
The ScriptEngineManager

- Auto-**discovers scripting languages**.
- Determines **which engine factory** to use.Gives you an **engine** by:
  - Language name.
  - Mime type.
  - File extension.
- It is a factory of factories.
- Returns an engine instance instead of the factory (for simplicity).



ScriptEngineFactory and ScriptEngine

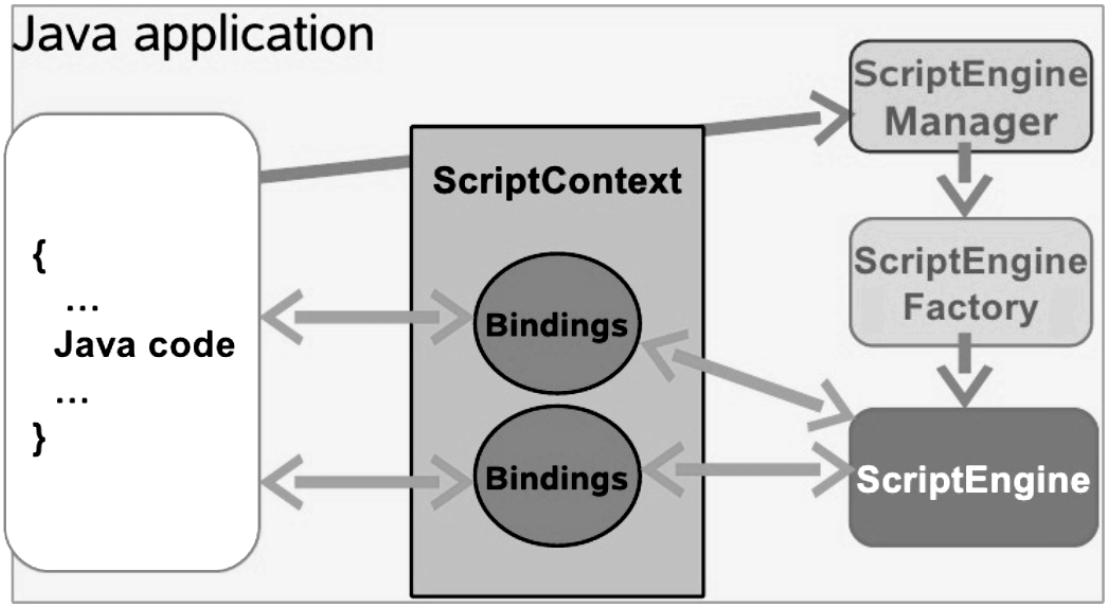
- **ScriptEngineFactory**
  - Maps mime types, file extensions, and language names to itself.
  - Produces instances of script engines for a certain language.
  - Provides syntax for method calls and output statements.
  - Provides mechanisms for building a script from statements.
- **ScriptEngine**
  - Interprets and runs (evaluates) your scripts.



  - Manages the context of variables and function names.
  - Might be Invocable and/or Compilable.

Bindings

- **Bindings** are mappings from names (strings) to objects.
  - Variable names to values/objects.
  - Function names to functions.
  - ...
- A **script context** is a collection of bindings with different scopes. It maps a scope to a set of bindings.
  - Globally bound names (global scope).
  - Locally bound names (engine-local scope).
  - ...



- ScriptEngine uses a ScriptContext for bindings.

Using Bindings I

- Every script variable is stored as a binding.
- E.g. add a local binding for the variable named k:

```
1 engine.eval("n = 17");
```

- In the next statement the variable is available:

```
1 Object result = engine.eval("n + 1");
```

- **Java objects** can be bound to script variables.
- Convenient method to bind a Java object to a script variable:

```
1 engine.put("k", 17) // Add a local binding.
2 engine.put("p", new Polynomial(2))
```

- The script can access all the bound variables, regardless of the type:

```
1 Object result = engine.eval("k + 1");
```

- Retrieve variables that were bound by scripting statements:

```
1 Object result = engine.get("n");
```

Using Bindings II

- Import Java packages in your script.
- Use a dialog from Java in your script:

```
1 jsEngine.eval("importPackage(javax.swing);" +
2     "JOptionPane.showMessageDialog(null,'Hello');");
```

- Example manipulate a java.util.List by a Script:

```
1 List<String> l = new ArrayList<String>();
2 l.add("Hello"); l.add("World");
3 engine.put("l", l); // Make l accessible as l
4 engine.eval("l.set(0,'Bye');" +
5     "l.toArray().map(println);");
```

- Manage your own bindings.
- Collect some bindings in an object of type Bindings.
- Pass the collected Bindings to the eval method.

```
1 Bindings scope = engine.createBindings();
2 scope.put("b1", new JButton()); scope.put(...
3 engine.eval(scriptString, scope);
```

Example – Calculator

- Writing a calculator.
  - Parsing mathematical expressions in infix notation.
  - Implementing trigonometric, hyperbolic, logarithmic, exp... functions.
  - Using variables for intermediate results.
  - Provide simple user-defined functions.
  - ...
- We need 4 lines of code:

```
1 ScriptEngine e = new ScriptEngineManager().getEngineByName("js");
2 String l = "'Simple Calculator'";
3 for (Scanner in= new Scanner(System.in); l.length()>0; l=in.nextLine())
4     System.out.println(e.eval("with (Math) { " + l + " }"));
```

- Put a try-catch block around line 4 to avoid exit on input error.

Calling Scripting Functions / Methods

- Script engine implements the **Invocable** interface.
  - This is an optional feature.
  - Rhino JavaScript engine is invocable.

```
1 if (engine implements Invocable) {
2     Invocable invEng = (Invocable) engine;
3     ...
4 }
```

- Invoking a function:

```
1 invEng.invokeFunction("fncName", arg1, arg2);
```

- Invoking a method:

```
1 invEng.invokeMethod(obj, "mtdName", arg1, arg2);
```

Using Interfaces I

- Ask the scripting engine to implement a Java interface.
  - Details depend on the scripting engine.
  - Typically: Supply a function for each method of the interface.
- E.g., consider the interface:

```
1 public interface Greeter {
2     String greet(String whom);
3 }
```

- Provide a script function greet:

```
1 invEng.eval("function greet(x) {return 'Hello '+x}");
```

- Ask the scripting engine to implement Greeter:

```
1 Greeter g = invEngine.getInterface(Greeter.class);
```

- Make a plain Java call to invoke the script function greet:

```
1 String result = g.greet("World");
```

Using Interfaces II

- In an object-oriented scripting language, you can access a script class through a matching Java interface.
- Consider the following prototype of MyGreeter:

```
1 function MyGreeter(text) { this.text = text; }
2 MyGreeter.prototype.greet = function(x) {
3     return this.text + x;
4 }
```

- Construct greeter objects with different salutations:

```
1 Object gObj = engine.eval("new MyGreeter('Hello ')");
```

- "Cast" the JavaScript object to a Java object implementing Greeter:

```
1 Greeter g=invEngine.getInterface(gObj, Greeter.class);
```

Compiling a Script

- Compile scripting code for repeated execution.
- Increases the efficiency of script execution.
  - This is an optional feature.
  - Rhino JavaScript engine can compile scripts.
- Engines implement the Compilable interface.
- Compile a script:

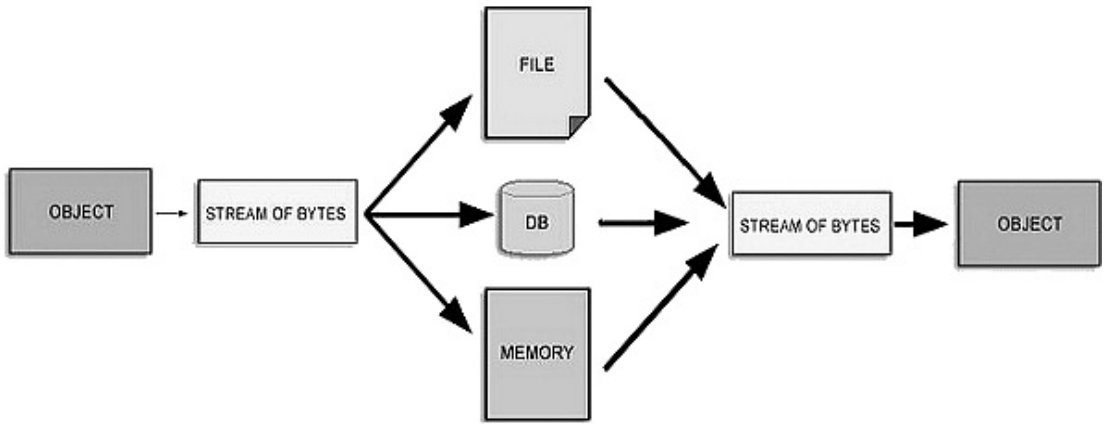
```
1 Reader reader = new FileReader("myscript.js");
2 CompiledScript script = null;
3 if (engine implements Compilable)
4     script = ((Compilable) engine).compile(reader);
```

- Once the script is compiled, you can execute it:

```
1 if (script != null)
2     script.eval();
3 else
4     engine.eval(reader);
```

Serialization

- Ability to store and retrieve Java objects.
- Serialization:
  - Represent state of object sufficient to reconstruct the object.
  - Maintain the relationships between objects.
  - When an object o is stored, all objects that are reachable from o are stored as well.
- Reconstructing the object is called deserialization.
- Use cases of serialization/deserialization:
  - Store and load the state of a program.
  - Create deep copies of an object.
  - ...
- Convert: Stream of bytes $\Leftrightarrow$ Object



Writing an Object

- Create a **java.io.ObjectOutputStream**.
- Use the method **writeObject**.
- Write a string object and an integer array into a file:

```
1 try (ObjectOutputStream oos = new ObjectOutputStream(
2         new FileOutputStream("test.out"))) {
3     oos.writeObject("Hello");
4     oos.writeObject(new int[] { 3, 4, 5 });
5 }
```

- Write into a byte array in the memory:

```
1 ByteArrayOutputStream b = new ByteArrayOutputStream();
2 try (ObjectOutputStream oos = new ObjectOutputStream(b)) {
3     oos.writeObject("Hello");
4     oos.writeObject(new int[] { 3, 4, 5 });
5 }
```

Read an Object

- Create a **java.io.ObjectInputStream**.
- Use the method **readObject**.
- Read a string object and an integer array from a file:

```
1 try (ObjectInputStream ois = new ObjectInputStream(
2         new FileInputStream("test.out"))) {
3     String state1 = (String) ois.readObject();
4     int[] state2 = (int[]) ois.readObject();
5     System.out.println(state1 + Arrays.toString(state2));
6 }
```

- Create a deep copy of an object by serialization:

```
1 ByteArrayOutputStream b = new ByteArrayOutputStream();
2 try (ObjectOutputStream oos = new ObjectOutputStream(b)) {
3     oos.writeObject(...);
4 }
5 try (ObjectInputStream ois = new ObjectInputStream(
6         new ByteArrayInputStream(b.toByteArray()))) {
7     copy = (...) ois.readObject();
8 }
```

The Interface Serializable

- Implement the interface **java.io.Serializable.**
- When you make a class serializable, all of its fields must be serializable as well.
  - All primitive types are serializable.
  - Many built-in objects are serializable:
    - All arrays are serializable.
    - All collections from java.util (ArrayList, HashMap, TreeSet,...)
    - String, Integer, Double, BigDecimal, URL, Date, Point, Random,...
  - But your own custom types might not be serializable!
- If you try to save an object that is not serializable or has non-serializable fields, you will get a **NotSerializableException**.

Copy Serializable Objects

- Use serialization to obtain a byte array form a serializable object.
  - Byte array contains the state information.
  - All the referenced objects are serialized too.
- Deserialize the byte array to obtain a deep copy.
  - All the referenced objects are copied.
  - No references to any “input object”.
- We can create a generic deepCopy method:

```
1  public static <T extends Serializable> T deepCopy(T toCopy) {
2      ByteArrayOutputStream b = new ByteArrayOutputStream();
3      try (ObjectOutputStream oos = new ObjectOutputStream(b)) {
4          oos.writeObject(toCopy);
5          ObjectInputStream ois = new ObjectInputStream(
6              new ByteArrayInputStream(b.toByteArray()));
7          return (T) ois.readObject();
8      } catch (Exception e) {
9          return null;
10     }
11 }
```

Example: Serializable Polynomial I

- Create a polynomial and store it in a file.

```
1  Polynomial p1 = new Polynomial(3);
2  p1.add(-7.0, 7,2,0);
3  p1.add(0.5, 3,2,2);
4  p1.add(2.0, 0,0,0);
5
6  try (ObjectOutputStream oos = new ObjectOutputStream(
7                       new FileOutputStream("poly.out"))) {
8      oos.writeObject(p1); // EXCEPTION
9  }
```

- java.io.NotSerializableException: at.jku.teaching.swtech.Polynomial.
- Make Polynomial.java serializable.
- Implement the interface java.io.Serializable.
- Make all the referenced objects serializable.
  - Anonymous implementations must be serializable too.

Example: Serializable Polynomial II

- The class Polynomial.java:

```
1  public class Polynomial {
2      private int numVars;
3      private TreeMap<int[], Double> monomials =
4          new TreeMap<>(new Comparator<int[]>() {
5              public int compare(int[] v, int[] w) {
6                  for (int i = 0; i < v.length; i++)
7                      if (w[i]-v[i] != 0) return w[i]-v[i];
8                  return 0;
9              }});
10     ...
```

- Quiz: What needs to be done?
- Make Polynomial and all the referenced objects serializable:

```
1  public class Polynomial implements Serializable {
2      private int numVars;
3      private static interface CompSer<T>
4                          extends Comparator<T>, Serializable {}
5      private TreeMap<int[], Double> monomials =
6          new TreeMap<>(new CompSer<int[]>() {
7              ...
```

- Anonymous implementations must be serializable too.

Versioning – serialVersionUID

- Versioning issue with serializing / deserializing objects.
  1. Save an object.
  2. Edit and recompile the class.
  3. Try to load the (now obsolete) object.
- Serializable objects should have a field inside named serialVersionUID that marks the "version" of the code.
  (If your class doesn't change, you can set it to 1 and never change it.)

```
1 public class Polynomial implements Serializable {
2     private static final long serialVersionUID = 1;
3     ...
```

Control the Serialization Output

- The **transient** keyword specifies that a field is not part of the persistent state of the object.
  - Temporary variables.
  - Variables that contain local information.
  - ...
- Transient fields are not saved during serialization.
- Static fields are not saved during serialization.
- You can override the default notion of how objects are serialized:
  - Make a field transient.
  - Define a different method to write/read the content:

```
1 private void writeObject(ObjectOutputStream out) throws ... {
2     out.defaultWriteObject(); // Default for NON-TRANSIENT
3     out.writeObject("Additional Information");
4 }
5 private void readObject(ObjectInputStream in) throws ... {
6     in.defaultReadObject();    // Default for NON-TRANSIENT
7     String additional = (String)in.readObject();
8 }
```

The Interface Externalizable

- Create your own format.
- By implementing the interface **java.io.Externalizable**.

```
1 public interface Externalizable extends Serializable {
2     void writeExternal(ObjectOutput out) throws ...;
3     void readExternal(ObjectInput in) throws ...;
4 }
```

- Complete control over the format of the stream for an object.
- Supersede customized implementation of writeObject readObject.
- Coordinate with the supertype to save its state.

JAXB for XML Serialization

- Java supports JAXB. (API package javax.xml...)
- Create a JAXBContext to read/write object from/to XML.
- JAXB treats pairs of Getter+Setter and public fields as mapped.
- The behavior can be changed by Annotations.
- You need to provide a mapping for the XML root element.
  - Annotate the root.
  - Create a JAXBElement to write the root and typed unmarshalling.
- JAXB needs a default constructor (or an adapter).
- Example: Define XML structure of Polynomial.java.

```
1  // Declare root element
2  @XmlRootElement public class Polynomial {
3      // Save numVars as attribute of the root element
4      @XmlAttribute private int numVars;
5      // Save monomials as child element nested into root
6      @XmlElement private TreeMap<int[], Double> monomials
         ...
7      // Provide default constructor
8      private Polynomial() {}
```

Using XML for Serialization I

- The interface Serializable is not needed.
- Create a JAXBContext to read/write Polynomial from/to XML.

```
1 Polynomial p1 = new Polynomial(3);
2 p1.add(-7.0, 7, 2, 0);
3 p1.add(0.5, 3, 2, 2);
4 p1.add(2.0, 0, 0, 0);
5
6 JAXBContext jc = JAXBContext.newInstance(p1.getClass());
```

- Write Polynomial to XML.

```
1 jc.createMarshaller().marshal(p1, new File("test.xml"));
```

- Read Polynomial from XML.

```
1 Polynomial p2 = (Polynomial) jc.createUnmarshaller()
2                              .unmarshal(new File("test.xml"));
```

Using XML for Serialization II

- Works out of the box for Java Beans:
  - Java Beans contain a zero-argument constructor.
  - Access to properties using getter and setter methods.
- Create a JAXBElement to write the root and use typed unmarshalling.
- E.g. Write and read the Java bean Person.java:

```
1  File data = new File("test.xml");
2  Person p =new Person("Muster","Max",new Date(99,SEPTEMBER,9));
3
4  jc = JAXBContext.newInstance(Person.class);
5  // Marsal with JAXBElement
6  jc.createMarshaller().marshal(
7      new JAXBElement<Person>(new QName("root"), Person.class, p), data);
8  // Typed Unmarshalling
9  p = jc.createUnmarshaller().unmarshal(
10     new StreamSource(data), Person.class).getValue();
```

Using XML for Serialization III

- Generic methods writeToXML/readFromXML for arbitrary Java bean.
- For convenience – easier to use:

```
1  Person p = readFromXML(Person.class, new File("test.xml"));
2  writeToXML(Person.class, p, new File("test.xml"));
```

```
1  public static <T> void writeToXML(Class<T> clazz, T javaBean,
2                                    File xmlOut) throws JAXBException {
3      JAXBContext jc = JAXBContext.newInstance(clazz);
4      jc.createMarshaller().marshal(new JAXBElement<T>(
5          new QName(clazz.getName()), clazz, javaBean), xmlOut);
6  }
7  public static <T> T readFromXML(Class<T> clazz, File xmlIn)
8                                                  throws JAXBException {
9      JAXBContext jc = JAXBContext.newInstance(clazz);
10     return jc.createUnmarshaller().unmarshal(
11         new StreamSource(xmlIn), clazz).getValue();
12 }
```