Effective Java Programming

controlling class loading

Structure

- controlling class loading
  - process of loading classes
  - when to implement your own classloader
  - greedy class loading
  - reduce number of classes

Process of loading classes

- classes are loaded through the classloader
- all classes are loaded dynamically
  - when a static member is used for the first time
    - classloader checks, if *Class* object already exists
    - if not – reads *.class* file
      - verifies the code
      - generates object of type *Class*
      - loads all dependencies (interfaces, super classes, class attributes)
    - class has to be loaded prior to creating an object
    - class can be loaded earlier
- a class can be omitted in current application run
  - no use cases accessed the class
  - only necessary classes get loaded

The *Class* class

- for every class used there is an object of type *Class*
  - created by the classloader during load
  - contains all informations about the class
    - name, canonnical name, type…
    - methods, attributes, annotations, constructors
  - creates all instances of given class
  - holds value of static attributes
- you can access the *Class* object
  - statically: ClassName.class
  - through the instance: object.getClass();
  - dynamically: Class.forName(„package.ClassName”)
    - dynamically creating an instance: newInstace();

When to implement own classloader

- sometimes we have specific requirements regarding loading classes
  - they are not in *classpath*
  - they have to downloaded from a server
  - *.class* files are encoded
  - classes are provided dynamically (i.e. servlet container)
- you can implement your own classloader
  - not very common
  - may lead to hard-to-find bugs
  - extend *ClassLoader*
    - override *findClass(String className)*
      - load code of given class
      - invoke *defineClass* to return control to VM

Own Classloader – example

```
class MyClassLoader extends ClassLoader {
  private byte[] myClassLoading(String name) throws
  ClassNotFoundException {
    // code for loading the class
  }

  @Override
  protected Class findClass(String name) throws
  ClassNotFoundException {
    byte[] classBytes = myClassLoading(name);
    int length = classBytes.length;
    Class clazz = defineClass(name, classBytes, 0, length);
    if (null == clazz ) {
      throw new ClassNotFoundException(name);
    }
    return clazz;
  }
}
```

Controll over loading classes

- to many classes can influence memory usage
- techniques reducing number of loaded classes exist
  - all techniques in this module are based on reflections
    - reflections are slower then normal method invocation
    - in many cases the benefits of using reflections exceed reduction in speed
  - cannot be widely used
    - designed to reduce influence of class loading on memory usage
    - have to be carefully used and profiled

Greedy class loading

- class has to be loaded before instantiating
- many factors can cause earlier class loading
  - i.e. some JIT compilers load all classes used by a method before compilation

```
public Translator getTranslator(String fileType) {
  if (fileType.equals(„doc”) {
    return new WorldTranslator();
  } else if (fileType.equals(„html”) {
    return new HTMLTranslator();
  } else if (fileType.equals(„txt”) {
    return new PlainTranslator();
  } else if (fileType.equals(„xml”) {
    return new XMLTranslator();
  } else ...
}
```

Controlling greedy class loading

```
public static Translator getTranslator(String fileType) throws
  Exception {
  Object result;
  if (fileType.equals(„doc”) {
    result = Class.forName(„WorldTranslator”).newInstance();
  } else if (fileType.equals(„html”) {
    result = Class.forName(„HTMLTranslator”).newInstance();
  } else if (fileType.equals(„txt”) {
    result = Class.forName(„PlainTranslator”).newInstance();
  } else if (fileType.equals(„xml”) {
    result = Class.forName(„XMLTranslator”).newInstance();
  } else ...
  return (Translator) result;
}
```

Reduce number of classes

- reducing number of classes will be described on listeners model
  - encourages to implement many little classes
  - can influence memory usage
- techniques
  - joining listeners
  - using reflection
  - using dynamic proxy

Simple inner classes

- inner classes have huge impact on memory usage
- hundred bytes big *class* file of the inner class can take up to 3KB in memory
- when using many listeners it can become a problem
- optimize

Simple inner classes

```
public class InnerClasses extends JFrame {
  ...
  class OpenAction implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      open();
    }
  }
  class CloseAction implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      close();
    }
  }
  class SaveAction implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      save();
    }
  }
}
```

Simple inner classes

- constructor for previous class

```
public InnerClasses() {
  JButton open = new JButton(„Open”);
  JButton close = new JButton(„Close”);
  JButton save = new JButton(„Save”);
  open.addActionListener(new OpenAction());
  close.addActionListener(new CloseAction());
  save.addActionListener(new SaveAction());
  ...
}
```

Joining listeners

- implement many listeners in one class
- join listeners of same type into one
- both solutions lead to:
  - reducing number of classes
  - reducing number of objects
    - 1 listener for more elements
- solution is also problematic
  - lead to *blob* and *spaghetti-code*
  - code based on name of elements
    - which can change and can need internationalization
  - hard to maintain

Joining listeners

```
class ButtonAction implements ActionListener {
  public void actionPerformed(ActionEvent e) {
    JButton b = (JButton)e.getSource();
    if (b.getText().equals(„Open”)) {
      open();
    } else if (b.getText().equals(„Close”)) {
      close();
    } else if (b.getText().equals(„Save”)) {
      save();
    }
  }
}
```

Joining listeners

- constructor after joining listeners

```
public InnerClasses() {
  JButton open = new JButton(„Open”);
  JButton close = new JButton(„Close”);
  JButton save = new JButton(„Save”);
  ActionListener listener = new ButtonAction();
  open.addActionListener(listener);
  close.addActionListener(listener);
  save.addActionListener(listener);
  ...
}
```

Use of reflection

- to avoid class loading use **reflections**
  - you can also avoid creating classes
- better maintenance then before
- problems:
  - no type verification during run
  - worse execution time
    - reflection is slower then direct calls
    - important in loops and algorithms
    - negligible in GUI

Use of reflection

```
class ReflectionAction implements ActionListener {
  private static Class[] argType = {};
  private static Object[] args = {};
  private String methodName;
  private Object target;
  // MethodName - enum
  public ReflectionAction(Object target, MethodName methodName){
    this.target = target;
    this.methodName = methodName.toString();
  }
  public void actionPerformed(ActionEvent e) {
    try {
      Method method = target.getClass().getMethod(methodName,
  argType);
      method.invoke(target, args);
    } catch (Exception ex) {}
  }
}
```

Use of reflection

- constructor with reflections

```
public InnerClasses() {
  JButton open = new JButton(„Open”);
  JButton close = new JButton(„Close”);
  JButton save = new JButton(„Save”);
  ActionListener listener = new ButtonAction();
  open.addActionListener(new ReflectionAction(this,
    MethodName.OPEN);
  close.addActionListener(new ReflectionAction(this,
    MethodName.CLOSE);
  save.addActionListener(new ReflectionAction(this,
    MethodName.SAVE);
  ...
}
```

Using dynamic proxy

- previous solution won't work when joining diferrent events of different listeners
- dynamic proxies come in hand
  - *java.lang.reflect.Proxy*
  - generate new class during runtime!
  - not for day-to-day use
    - complicated implementation
  - recommended for flexible applications

Using dynamic proxy

```
class MyProxy implements InvocationHandler {
  private String methodName; // method to invoke
  private Object target; // object to use

  public static Object makePorxy(Object target, String
  methodName, Class impl){
    MyProxy myProxy = new MyProxy();
    myProxy.target = target;
    myProxy.methodName = methodName;
    ClassLoader loader = target.getClass().getClassLoader();
    // create a new proxy, which will implement the interface
    // impl. the object will forward all calls to the proxy
    // through invoke method in myProxy
    return Proxy.newProxyInstance(loader, new Class[]{impl},
  myProxy);
  }
```

Using dynamic proxy

```
  @Override
  public Object invoke(Object proxy, Method method, Object[]
  args) {
    try {
      Object[] noArgs = {};
      Class[] argTypes = {};
      // we are ignoring arguments
      Method targetMethod = target.getClass()
       .getMethod(methodName, argTypes);
      return targetMethod.invoke(target, noArgs);
    } catch (Exception ex) {
      return null;
    }
  } // end of method
} // end of class
```

Using dynamic proxy

- constructor with dynamic proxy

```
public InnerClasses() {
  JButton open = new JButton(„Open”);
  JButton close = new JButton(„Close”);
  JButton save = new JButton(„Save”);
  ActionListener listener = new ButtonAction();
  open.addActionListener((ActionListener)MyProxy.makeProxy(this,
  „open”, ActionListener.class));
  close.addActionListener((ActionListener)MyProxy.makeProxy(
    this, „close”, ActionListener.class));
  save.addActionListener((ActionListener)MyProxy.makeProxy(this,
  „save”, ActionListener.class));
  ...
}
```

Is it worth using

- some of the techniques might be frightening
- these solutions are not only used with listeners
- early versions of Swing loaded too many classes
  - provided techniques eliminated this problem by delaying class loading until really necessary
  - reflections used in *UIDefualts* and *CellEditors*
    - in some applications it eliminated loading > 200 classes
    - critically important for minimalistic solutions for devices with limited resources

Conclusions

- what are the methods for restricting class loading?
- what are their benefits and dangers?