61FIT3JSD
Fall 2022

Lecture 5

Meta programming

Lecture outline

- Meta-programming
  - Reflection
  - Generics
  - Annotation
  - Meta-programming application

What is meta-programming?

- A term used to refer to program manipulation by another program
- Meta-programming as a language feature:
  - subject and object programs are written in the same language
- Java supports meta-programming through:
  - **Reflection**: operate on class and its members
  - **Generics**: further generalise code
  - **Annotation**: define metadata for class and its members

Reflection

- Enables examination or modification of the run-time behaviour of a program
- Three common usages:
  - extensibility: run-time instantiation of external classes
  - browser & visual development: member listing & better quality code
  - debugger and test tools: run-time state and test API discovery

About classes

- Every class has a `java.lang.Class` object created for it when loaded
- `java.lang.Class`: the entry point for the reflection API
  - provides methods to examine class information and object state
- Three methods to obtain the `Class` object:
  - `Object.getClass()`
  - The `class` member
  - `Class.forName()`

Object.getClass

- Invoked on an object of a class
- Works only for object types
- Examples:

```
Class c = "foo".getClass(); // java.lang.String
c = System.out.getClass(); // java.io.PrintStream
byte[] bytes = new byte[1024];
c = bytes.getClass(); // [B
```

The `class` member

- Invoked directly on the class
- Used when no objects are available
  - works also for primitive types
- Examples:

```
c = boolean.class; // boolean

c = int[][][].class; // [[[I

c = java.io.PrintStream.class; //
java.io.PrintStream

c = Customer.class; // Customer
```

Class.forName (1)

- Invoked with a fully qualified class name as arg
- Usually used for external (unloaded) classes
- Only works for object types
- Examples:

```
c = Class.forName("courseman.Student");
c = Class.forName("[D"); // double[].class
c = Class.forName("[[Ljava.lang.String;");
// String[][].class
```

Use with try...catch

```
try {
   c = Class.forName("courseman.Student");
   c = Class.forName("[D");
   c = Class.forName("[[Ljava.lang.String;");
} catch (ClassNotFoundException e) {
   e.printStackTrace();
}
```

Access information about a class

- Via methods of a `Class` object
- Retrieve information about:
  - other classes: super, inner, outer classes
  - fields: instance variables
  - methods
- Defined in `java.lang.reflect` package:
  - `Field` → fields
  - `Method` → methods
  - `Constructor` → constructors

Other classes

- `getSuperClass()`: returns the super class
  - works for all
- `getClasses()`: returns all *public* classes, interfaces, and enums
  - if this class is outer
- `getDeclaredClasses()`: all classes/interfaces/enums *declared* in this class
  - if this class is outer
- `getEnclosingClass()`: returns the outer class
  - if this class is inner

Field

- Class `java.lang.reflect.Field`
- A field has type and value
- Methods:
  - `getType()`: return the declared type
  - Setters for field value

FieldSpy

`lect04.meta.FieldSpy`

Method

- Class: `java.lang.reflect.Method`
- Has two sets of methods:
  - Getters for method definitions (and others)
  - Invocation of a method

Constructor

- Class: `java.lang.reflect.Constructor`
- Provides two sets of methods:
  - Getters for constructor definitions
  - Creating a new class instance

Obtaining constructor information

- Methods:
  - `getDeclaredConstructors()`: obtain all declared constructors
  - `getParameterTypes`: obtain the parameter list as `Class[]` array

ConstructorSift

`lect04.meta.ConstructorSift`

Create a new instance

- Two methods:
  - invoke `newInstance` on the class object
  - invoke `newInstance` on a constructor object of the class
- `Class.newInstance()`:
  - invokes the default constructor of the class
- `Constructor.newInstance(...)`:
  - can invoke constructors other than the default
  - takes `Object[]` as arguments

ConstructorDemo (1)

```
import static java.lang.System.out;

class A {
    A() {
        out.println("A.init()");
    }

    public A(String s) {
        out.println("A.init(" + s + ")");
    }
}
```

...

```
public static void main(String[] args) {
    Constructor c;
    A a;
    try {
        c = A.class.getDeclaredConstructor(null);
        a = (A) c.newInstance();
        c = A.class.getConstructor(String.class);
        a = (A) c.newInstance("Hello world");

    } catch (...) {...}
}
```

Drawbacks of reflection

- Performance overhead
- Security restrictions
- Exposure of internals

Avoid if alternatives exist !

Generics

- Provides an abstraction over types:
  - types that can take a type variable as parameter
- Benefits:
  - improved code readability (no casting)
  - robustness (type safety)

Example

- Non-generic code:

```
List l = new ArrayList();
l.add(123);  // auto-boxing
Integer o = (Integer) l.get(0);  //casting
```

- Generic code:

```
List<Integer> l = new ArrayList<Integer>();
l.add(123);  // auto-boxing
Integer o = l.get(0);        // no casting
```

A generic class example

- Takes type variable E as a parameter
- E is set to an actual type at run-time

```
public class SimpleGeneric<E> {
    void print(E x) {
        if (x != null) {
            System.out.println(x);
        } else {
            System.out.println("null");
        }
    }
}
```

Generic collection

```
public class SimpleGenericCollection<E>
                                    extends SimpleGeneric {
    private List<E> list;

    public SimpleGenericCollection() {
        super();
        list = new ArrayList<E>();
    }

    public void add(E x) {
        list.add(x);
    }
```

Generic method

- Method definitions using type variables as parameters
- A higher level of abstraction than polymorphic methods:
  - types are not fixed

Generic method example

```
public class Utility {
    public static <T> T getMidpoint(T[] a) {
        return a[a.length / 2];
    }
    public static <T> T getFirst(T[] a) {
        return a[0];
    }
    public static void main(String[] args) {
        String[] b = {"A", "B", "C"};
        Double[] c = {1.0, 1.5, 2.0};
        String midString = Utility.<String>getMidpoint(b);
        double firstNumber = Utility.<Double>getFirst(c);
    }
}
```

Annotation

- A metadata that describes a program's features for other programs to process
- Methods and/or fields of interest are annotated
- Java supports:
  - definition of new annotation types
  - using annotation types in the code
- Used together with Java's reflection to provide more powerful capabilities

Annotation example

```
public class Course {
    @DomainConstraint(type      =   "Integer",
                      min       =   1)
    private int id;

    @DomainConstraint(type      =   "String",
                      optional  =   false,
                      length    =   10)
    private String name;
}
```

Defining an annotation type

- Similar to normal interface declarations
- An at-sign (@) precedes the `interface` keyword
- A method definition:
  - defines an annotation element
  - must have no have parameters
  - return type must be primitive, `String`, `Class`, `enum`s, annotations, or an array of these
  - may use `default` keyword to set a default value

Example: DomainConstraint

```
@Retention(RetentionPolicy.RUNTIME)
public @interface DomainConstraint {
  public String type();
  public boolean mutable() default true;
  public boolean optional() default true;
  public int length() default  -1;
  public double min() default Double.NaN;
  public double max() default Double.NaN;
}
```

Using an annotation

@Annotation_Name(Element_Value_Pairs)

- *Element_Value_Pairs* may be omitted
- Used as a special modifier
- Precedes other modifiers (by convention)
- Example:

```
@DomainConstraint(type="String",
                  optional=false,length=50)
private String name;
```

A meta-programming application

- Objectives:
  - Validate input data values for manipulating objects
  - Automatically create new objects of a **domain class**
- Design:
  - **Annotate** fields of a domain class with `DomainConstraint`
  - Use **reflection** to:
    - obtain fields and domain constraints
    - create new objects (using `newInstance`)
  - Check input data value of a field against its domain constraint

Code

- `lect05.app`
  - `Course`: a domain class
  - `MetaApp`: implements the meta-programming tasks

Domain class

`lect05.app.Course`

Obtain fields and their DCs

```
List _fields = new ArrayList();
...
List fields = new ArrayList();
for (Iterator it = _fields.iterator();
                          it.hasNext();) {
  f = (Field) it.next();
  if (f.getAnnotation(dcls) != null) {
    fields.add(f);
  }
}
...
```

Check input data value against DC

```
public static Object validate(Object
value, DomainConstraint dc)
        throws NotPossibleException {
  ...
  // validate optional constraint
  if (!dc.optional() && value == null) {
      throw new NotPossibleException("
       validate: value cannot be null");
  }
```

...

```
...
// check and convert value
String type = dc.type();
if (type.equals("String")) {
   // validate length constraint
   if (dc.length() > 0) {
      ...
   }
}
```

...

```
if (type.equals("Integer") ||
    type.equals("Long") ||
    type.equals("Float") ||
    type.equals("Double")) {
  // convert value to number
  ...
  val = Integer.parseInt(value.toString());
  ...
```

...

```
  // validate min and max constraints
  if (dc.min() != Double.NaN) {
   if (((Number) val).doubleValue() < dc.min())
      ...
  }
  if (dc.max() != Double.NaN) {
   if (((Number) val).doubleValue() < dc.max())
   ...
  }
}
```

Other applications

- Automatically build data capturing functions
- Automatically generate database schema from domain classes
- Schema matching:
  - matches classes that have similar domain attributes

Summary

- Java supports meta programming via: reflection, generics, and annotation
- Reflection is powerful but requires care
- Generics provides abstraction over types
- Annotation provides metadata about program components