

# Programming Language Concepts: Lecture 10

Madhavan Mukund

Chennai Mathematical Institute

`madhavan@cmi.ac.in`

`http://www.cmi.ac.in/~madhavan/courses/pl2009`

PLC 2009, Lecture 10, 16 February 2009

# Reflection

Wikipedia says

*Reflection is the process by which a computer program can observe and modify its own structure and behaviour.*

# Reflection

Wikipedia says

*Reflection is the process by which a computer program can observe and modify its own structure and behaviour.*

Two components involved in reflection

- ▶ **Introspection**

A program can observe, and therefore reason about its own state.

# Reflection

Wikipedia says

*Reflection is the process by which a computer program can observe and modify its own structure and behaviour.*

Two components involved in reflection

- ▶ **Introspection**

A program can observe, and therefore reason about its own state.

- ▶ **Intercession**

A program can modify its execution state or alter its own interpretation or meaning.

# Reflection in Java

## Simple example of introspection

```
Employee e = new Manager(...);  
...  
if (e instanceof Manager){  
    ...  
}
```

# Reflection in Java

## Simple example of introspection

```
Employee e = new Manager(...);  
...  
if (e instanceof Manager){  
    ...  
}
```

What if we don't know the type that we want to check in advance?

# Reflection in Java

## Simple example of introspection

```
Employee e = new Manager(...);  
...  
if (e instanceof Manager){  
    ...  
}
```

What if we don't know the type that we want to check in advance?

Suppose we want to write a function to check if two different objects are both instances of the same class?

```
public static boolean classequal(Object o1, Object o2){  
    ...  
    // return true iff o1 and o2 point to objects of same type  
    ...  
}
```

# Reflection in Java ...

```
public static boolean classequal(Object o1, Object o2){...}
```

- ▶ Can't use `instanceof`
  - ▶ Will have to check across all defined classes
  - ▶ This is not even a fixed set!
- ▶ Can't use generic type variables
  - ▶ The following code is syntactically disallowed

```
if (o1 instanceof T) { ...}
```



# Introspection in Java

Can extract the class of an object using `getClass()`

- ▶ Import package `java.lang.reflect`

# Introspection in Java

Can extract the class of an object using `getClass()`

- ▶ Import package `java.lang.reflect`

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

class MyReflectionClass{
    ...
    public static boolean classequal(Object o1, Object o2){
        return (o1.getClass() == o2.getClass());
    }
}
```

# Introspection in Java

Can extract the class of an object using `getClass()`

- ▶ Import package `java.lang.reflect`

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

class MyReflectionClass{
    ...
    public static boolean classequal(Object o1, Object o2){
        return (o1.getClass() == o2.getClass());
    }
}
```

What does `getClass()` return?

# Introspection in Java

Can extract the class of an object using `getClass()`

- ▶ Import package `java.lang.reflect`

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

class MyReflectionClass{
    ...
    public static boolean classequal(Object o1, Object o2){
        return (o1.getClass() == o2.getClass());
    }
}
```

What does `getClass()` return?

- ▶ An object of type `Class` that encodes class information

# The class `Class`

A version of `classsequal` the explicitly uses this fact

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

class MyReflectionClass{
    ...
    public static boolean classequal(Object o1, Object o2){
        Class c1, c2;
        c1 = o1.getClass();
        c2 = o2.getClass();
        return (c1 == c2);
    }
}
```

# The class `Class`

A version of `classsequal` the explicitly uses this fact

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

class MyReflectionClass{
    ...
    public static boolean classsequal(Object o1, Object o2){
        Class c1, c2;
        c1 = o1.getClass();
        c2 = o2.getClass();
        return (c1 == c2);
    }
}
```

- For each currently loaded class `C`, Java creates an object of type `Class` with information about `C`

# The class `Class`

A version of `classEqual` that explicitly uses this fact

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

class MyReflectionClass{
    ...
    public static boolean classEqual(Object o1, Object o2){
        Class c1, c2;
        c1 = o1.getClass();
        c2 = o2.getClass();
        return (c1 == c2);
    }
}
```

- ▶ For each currently loaded class `C`, Java creates an object of type `Class` with information about `C`
- ▶ Encoding execution state as data — **reification**
  - ▶ Representing an abstract idea in a concrete form

# Using the `Class` object

Can create new instances of a class at runtime

```
...  
Class c = obj.getClass();  
Object o = c.newInstance();  
    // Create a new object of same type as obj  
...
```



# Using the `Class` object

Can create new instances of a class at runtime

```
...  
Class c = obj.getClass();  
Object o = c.newInstance();  
    // Create a new object of same type as obj  
...
```

Can also get hold of the class object using the name of the class

```
...  
String s = "Manager".  
Class c = Class.forName(s);  
Object o = c.newInstance();  
...
```

# Using the `Class` object

Can create new instances of a class at runtime

```
...
Class c = obj.getClass();
Object o = c.newInstance();
    // Create a new object of same type as obj
...
```

Can also get hold of the class object using the name of the class

```
...
String s = "Manager".
Class c = Class.forName(s);
Object o = c.newInstance();
...
```

..., or, more compactly

```
...
Object o = Class.forName("Manager").newInstance();
...
```

# The class `Class` ...

- ▶ From the `Class` object for class `C`, we can extract details about constructors, methods and fields of `C`

# The class `Class` ...

- ▶ From the `Class` object for class `C`, we can extract details about constructors, methods and fields of `C`
- ▶ Constructors, methods and fields themselves have structure
  - ▶ Constructors: arguments
  - ▶ Methods : arguments and return type
  - ▶ All three: modifiers `static`, `private` etc

# The class `Class` ...

- ▶ From the `Class` object for class `C`, we can extract details about constructors, methods and fields of `C`
- ▶ Constructors, methods and fields themselves have structure
  - ▶ Constructors: arguments
  - ▶ Methods : arguments and return type
  - ▶ All three: modifiers `static`, `private` etc
- ▶ Additional classes `Constructor`, `Method`, `Field`

# The class `Class` ...

- ▶ From the `Class` object for class `C`, we can extract details about constructors, methods and fields of `C`
- ▶ Constructors, methods and fields themselves have structure
  - ▶ Constructors: arguments
  - ▶ Methods : arguments and return type
  - ▶ All three: modifiers `static`, `private` etc
- ▶ Additional classes `Constructor`, `Method`, `Field`
- ▶ Use `getConstructors()`, `getMethods()` and `getFields()` to obtain constructors, methods and fields of `C` in an array.

# The class `Class` ...

Extracting information about constructors, methods and fields

```
...  
Class c = obj.getClass();  
Constructor[] constructors = c.getConstructors();  
Method[] methods = c.getMethods();  
Field[] fields = c.getFields();  
...
```

# The class `Class` ...

Extracting information about constructors, methods and fields

```
...  
Class c = obj.getClass();  
Constructor[] constructors = c.getConstructors();  
Method[] methods = c.getMethods();  
Field[] fields = c.getFields();  
...
```

`Constructor`, `Method`, `Field` in turn have functions to get further details



# The class `Class` ...

Example: Get the list of parameters for each constructor

```
...
Class c = obj.getClass();
Constructor[] constructors = c.getConstructors();
for (int i = 0; i < constructors.length; i++){
    Class params[] = constructors[i].getParameterTypes();
    ..
}
```

Each parameter list is a list of types

- ▶ Return value is an array of type `Class[]`

# The class `Class` ...

We can also invoke methods and examine/set values of fields.

```
...
Class c = obj.getClass();
..
Method[] methods = c.getMethods();
Object[] args = { ... }
    // construct an array of arguments
methods[3].invoke(obj,args);
    // invoke methods[3] on obj with arguments args
...
```

# The class `Class` ...

We can also invoke methods and examine/set values of fields.

```
...
Class c = obj.getClass();
..
Method[] methods = c.getMethods();
Object[] args = { ... }
    // construct an array of arguments
methods[3].invoke(obj,args);
    // invoke methods[3] on obj with arguments args
...

Field[] fields = c.getFields();
Object o = fields[2].get(obj);
    // get the value of fields[2] from obj
...
fields[3].set(obj,value);
    // set the value of fields[3] in obj to value
...
```

# Reflection and security

- ▶ Can we extract information about private methods, fields, ...?
- ▶ `getConstructors()`, ... only return publicly defined values
- ▶ Separate functions to also include private components
  - ▶ `getDeclaredConstructors()`
  - ▶ `getDeclaredMethods()`
  - ▶ `getDeclaredFields()`
- ▶ Should this be allowed to all programs?
- ▶ Security issue!
- ▶ Access to private components may be restricted

# Using reflection

- ▶ [BlueJ](#), a programming environment to learn Java

# Using reflection

- ▶ BlueJ, a programming environment to learn Java
- ▶ Can define and compile Java classes

# Using reflection

- ▶ [BlueJ](#), a programming environment to learn Java
- ▶ Can define and compile Java classes
- ▶ For compiled code, create object, invoke methods, examine state

# Using reflection

- ▶ [BlueJ](#), a programming environment to learn Java
- ▶ Can define and compile Java classes
- ▶ For compiled code, create object, invoke methods, examine state
- ▶ Uses reflective capabilities of Java — [BlueJ](#) need not internally maintain “debugging” information about each class



# Using reflection

- ▶ **BlueJ**, a programming environment to learn Java
- ▶ Can define and compile Java classes
- ▶ For compiled code, create object, invoke methods, examine state
- ▶ Uses reflective capabilities of Java — **BlueJ** need not internally maintain “debugging” information about each class
- ▶ Look up <http://www.bluej.org>

# Limitations of Java reflection

- ▶ Cannot create or modify classes at run time

- ▶ The following is not possible

```
Class c = new Class(...);
```

- ▶ Note that **BlueJ** must invoke Java compiler before you can use a new class

# Limitations of Java reflection

- ▶ Cannot create or modify classes at run time

- ▶ The following is not possible

```
Class c = new Class(...);
```

- ▶ Note that **BlueJ** must invoke Java compiler before you can use a new class
- ▶ Languages such as **Smalltalk** allow redefining methods at run time