LINGI2252 – PROF. KIM MENS

SOFTWARE MAINTENANCE & EVOLUTION

LINGI2252 – PROF. KIM MENS

REFLECTION (IN JAVA)

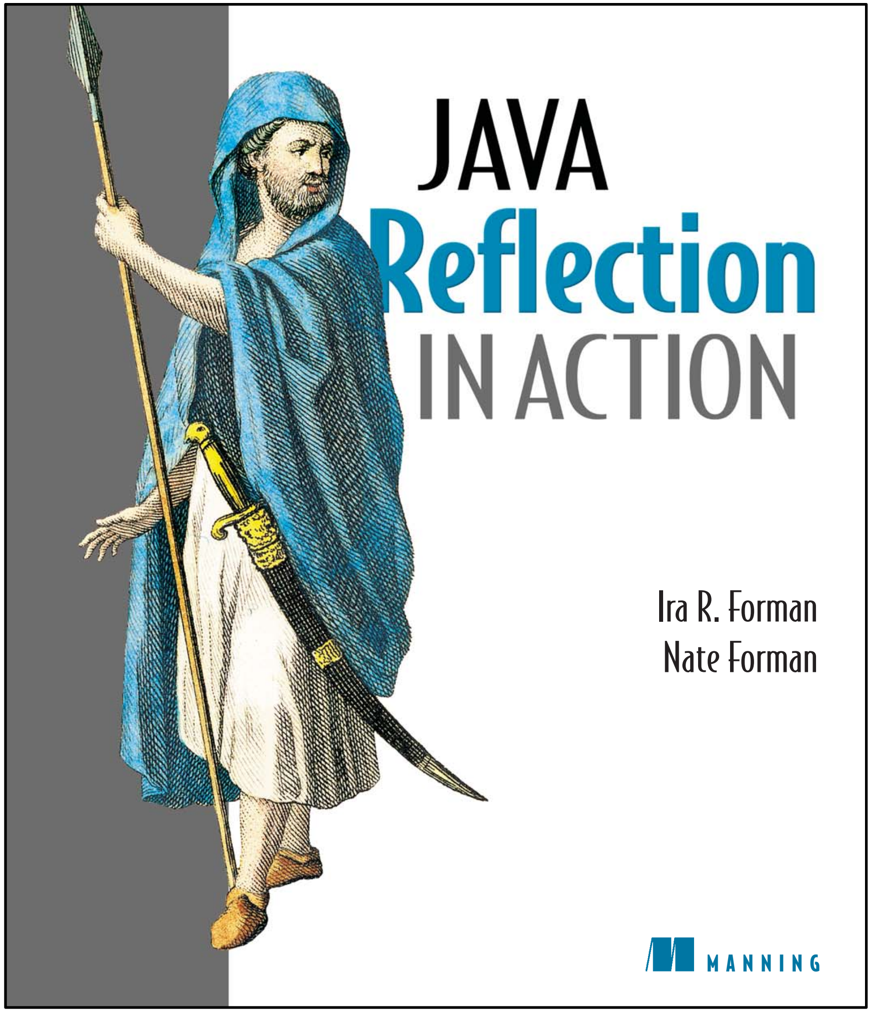
Lecture 10 a

Basics of Reflection in Java

Partly inspired on :

- Book “Java Reflection in Action”
- The Java Tutorials
- Slides by Prof. Walter Cazzola

Some References



ORACLE Java Documentation

The Java™ Tutorials

« Previous • Trail • Next »

Trail: The Reflection API

Uses of Reflection

Reflection is commonly used by programs which require the ability to exami
This is a relatively advanced feature and should be used only by developers
mind, reflection is a powerful technique and can enable applications to perfc

Extensibility Features
An application may make use of external, user-defined classes by cr
Class Browsers and Visual Development Environments
A class browser needs to be able to enumerate the members of clas
Information available in reflection to aid the developer in writing corre
Debuggers and Test Tools
Debuggers need to be able to examine private members on classes.
APIs defined on a class, to insure a high level of code coverage in a

Some Terminology

- **Computational reflection**: the ability of a running program to inspect and change itself.
- **Reification**: making domain/language entities (meta-level) accessible to a program (base level), so that it can be manipulated by computation.
- **Introspection**: self-examination; “read only” access to reified entities.
- **Intercession**: when you actually intervene in the program execution as well by manipulating the reified entities.

10 a.1 About Reflection in Java

About Reflection in Java

Java provides an API for reflection

- TRAIL: The Java Reflection API
- a rich set of operations for **introspection**
- some **intercession** capabilities

Can be used to examine or modify the runtime behaviour of applications running in the JVM

About Reflection in Java

Warning :

Reflection is an advanced feature

- should be used only by developers who have a strong grasp of the fundamentals of the language

When to avoid it ?

- in performance-sensitive applications
- in security-related code
- may expose some internals (e.g. accessing of private fields)

Example of a Reflective Program

- User interface implementation
- Contains several visual components (classes)
  - some developed in-house
  - some open source
  - some belong the standard Java libraries
  - some are bought
- All these components understand
  - setColor(Color aColor)
- Problem: how to invoke this method?
  - the components share no common supertype of interface;
  - only common base class is Object

(Taken from Chapter 1 of the book "Java Reflection in Action" by Forman & Forman)

Example of a Reflective Program


```
public class Component1 {

    Color myColor;

    public void setColor(Color color) {
        myColor = color;
    }

  ...
}
```

```
Component3 {

lor myColor;

blic void setColor(Color color) {
        myColor = color;
```

```
public class Component4

    Color myColor;

    public void setColor(C
        myColor = color;
    }

    ...
}
```

```
public class Component2 {

    Color myColor;

    public void setColor(Color color) {
        myColor = color;
    }

    ...
}
```

What's The Problem?


```
public class Main {

    static Object[] components = new Object[10];
    static Color color = new Color(0);

    public static void initializeComponents() {
        components[0] = new Component1();
        components[1] = new Component2();
        ...                             }

    public static void main(String[] args) {
        initializeComponents();
        for (int i = 0; i < args.length; i++)
                if (components[i] != null)
                    components[i].setColor(color);

    }
}
```

The method setColor(color) is undefined for the type Object

Possible Solutions?

Define setColor on Object

not a nice solution; not possible anyway

Use a typecast

impossible: Object is the only common supertype

Make components implement common interface

would work : we can use that interface as common type

unfortunately some code is not ours: cannot be changed

Use an adapter for each component

i.e. a “fake” component that delegates to the real one

all adapters implement a common interface

works, but explosion of extra classes and objects

More Solutions?

Use instanceOf and casting

```
if (components[i] instanceof Component1)
        ((Component1) components[i]).setColor(color);
  if (components[i] instanceof Component2)
        ((Component2) components[i]).setColor(color);
  ...
```

works, but code bloated with conditionals and calls

and concrete types hard-coded in the code

Why not use reflection?

to find the right setColor method to call and invoke it using reflection

Reflective Solution


```
public class Main {

    static Object[] components = new Object[10];
    static Color color = new Color(0);

    public static void initializeComponents() {
        components[0] = new Component1();
        components[1] = new Component2();
        ...                            }

    public static void main(String[] args) {
        initializeComponents();
        for (int i = 0; i < args.length; i++)
                if (components[i] != null)
                    setObjectColor(components[i],color);

    }
}
```

Reflective method

Reflective Solution


```
public static void setObjectColor( Object obj, Color color )   {
   Class cls = obj.getClass();
   try {
         Method method = cls.getMethod("setColor", new Class[] {Color.class} );
      method.invoke( obj, new Object[] {color} );
   }
   catch (NoSuchMethodException ex) {
      throw new IllegalArgumentException(
         cls.getName() + " does not support method setColor(Color)”);
   }
   catch (IllegalAccessException ex) {
      throw new IllegalArgumentException(
         "Insufficient access permissions to call"
         + "setColor(:Color) in class " + cls.getName());
   }
   catch (InvocationTargetException ex) {
      throw new RuntimeException(ex);
   }
}
```

Reflective Solution


```
public static void setObjectColor( Object obj, Color color )   {
   Class cls = obj.getClass();
   try {
         Method method = cls.getMethod( "setColor", new Class[] {Color.class} );
      method.invoke( obj, new Object[] {color} );
   }
   catch (NoSuchMethodException ex) {
      throw new IllegalArgumentException(
         cls.getName() + " does not support method setColor(Color)");
   }
   catch (IllegalAccessException ex) {
      throw new IllegalArgumentException(
         "Insufficient access permissions to call"
         + "setColor(:Color) in class " + cls.getName());
   }
   catch (InvocationTargetException ex) {
      throw new RuntimeException(ex);
   }
}
```

Query `obj` for its class

Reflective Solution


```
public static void setObjectColor( Object obj, Color color )   {
  Class cls = obj.getClass();
  try {
        Method method = cls.getMethod("setColor", new Class[] {Color.class} );
     method.invoke( obj, new Object[] {color} );
  }
  catch (NoSuchMethodException ex) {
     throw new IllegalArgumentException(
        cls.getName() + " does not support method setColor(Color)");
  }
  catch (IllegalAccessException ex) {
     throw new IllegalArgumentException(
        "Insufficient access permissions to call"
        + "setColor(:Color) in class " + cls.getName());
  }
  catch (InvocationTargetException ex) {
     throw new RuntimeException(ex);
  }
}
```

Query class object for its `setColor` method

Reflective Solution


```
public static void setObjectColor( Object obj, Color color )  {
  Class cls = obj.getClass();
  try {
      Method method = cls.getMethod("setColor", new Class[] {Color.class} );
    method.invoke( obj, new Object[] {color} );
  }
  catch (NoSuchMethodException ex) {
    throw new IllegalArgumentException(
      cls.getName() + " does not support method setColor(Color)");
  }
  catch (IllegalAccessException ex) {
    throw new IllegalArgumentException(
      "Insufficient access permissions to call"
      + "setColor(:Color) in class " + cls.getName());
  }
  catch (InvocationTargetException ex) {
    throw new RuntimeException(ex);
  }
}
```

Call resulting method on target `obj`

Reflective Solution


```
public static void setObjectColor( Object obj, Color color )   {
   Class cls = obj.getClass();
   try {
         Method method = cls.getMethod("setColor", new Class[] {Color.class} );
      method.invoke( obj, new Object[] {color} );
   }
   catch (NoSuchMethodException ex) {
      throw new IllegalArgumentException(
         cls.getName() + " does not support method setColor(Color)");
   }
   catch (IllegalAccessException ex) {
      throw new IllegalArgumentException(
         "Insufficient access permissions to call"
         + "setColor(:Color) in class " + cls.getName());
   }
   catch (InvocationTargetException ex) {
      throw new RuntimeException(ex);
   }
}
```

Catch error if class `obj` does not support `setColor` method

Reflective Solution

```
public static void setObjectColor( Object obj, Color color )  {
  Class cls = obj.getClass();
  try {
       Method method = cls.getMethod("setColor", new Class[] {Color.class} );
    method.invoke( obj, new Object[] {color} );
  }
  catch (NoSuchMethodException ex) {
    throw new IllegalArgumentException(
      cls.getName() + " does not support method setColor(Color)”);
  }
  catch (IllegalAccessException ex) {
    throw new IllegalArgumentException(
      "Insufficient access permissions to call"
      + "setColor(:Color) in class " + cls.getName());
  }
  catch (InvocationTargetException ex) {
    throw new RuntimeException(ex);
  }
}
```

Catch error if invoker cannot call `setColor` method

Reflective Solution


```
public static void setObjectColor( Object obj, Color color )   {
   Class cls = obj.getClass();
   try {
         Method method = cls.getMethod("setColor", new Class[] {Color.class} );
      method.invoke( obj, new Object[] {color} );
   }
   catch (NoSuchMethodException ex) {
      throw new IllegalArgumentException(
         cls.getName() + " does not support method setColor(Color)”);
   }
   catch (IllegalAccessException ex) {
      throw new IllegalArgumentException(
         "Insufficient access permissions to call"
         + "setColor(:Color) in class " + cls.getName());
   }
   catch (InvocationTargetException ex) {
      throw new RuntimeException(ex);
   }
}
```

Catch error if method `setColor` throws exception

Example of a Reflective Program

Introspection

- to query an object for its class
- to query a class for its methods

Dynamic invocation

- to dynamically call a method at run-time
  without specifying which one at compile-time

Care should be taken to handle all exceptions

This solution is

- flexible and elegant
- though somewhat verbose
- but has some performance penalties

10 a.2

The Java Meta-Object Protocol

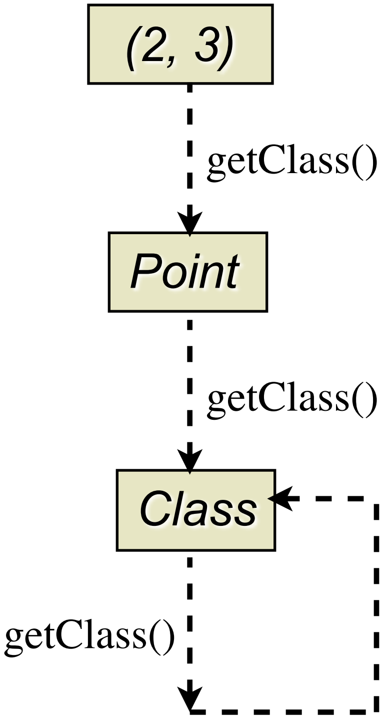
The Java Meta-Object Protocol


```
Point aPoint = new Point(2,3)
```

```
Class aClass = aPoint.getClass()
    ➡ class Point
```

```
Class aMeta = aClass.getClass()
    ➡ class java.lang.Class
```

```
Class aMeta2 = aMeta.getClass()
    ➡ class java.lang.Class
```



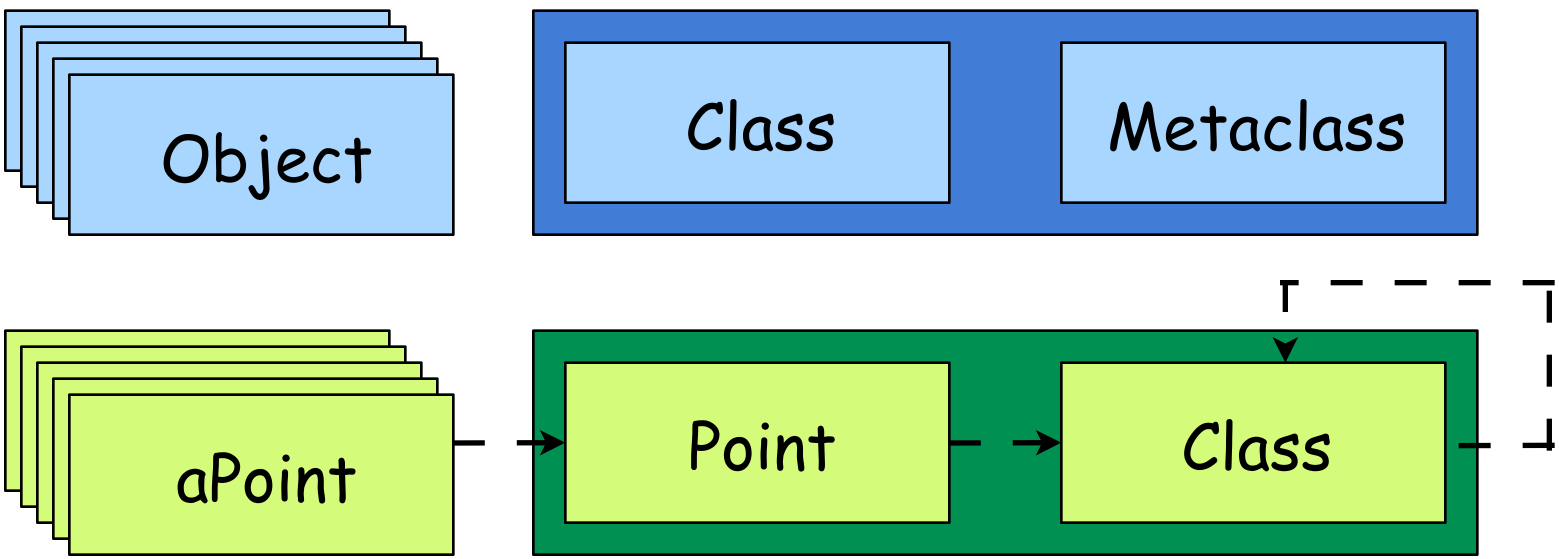
The Metaclass Loop in Java

instance of

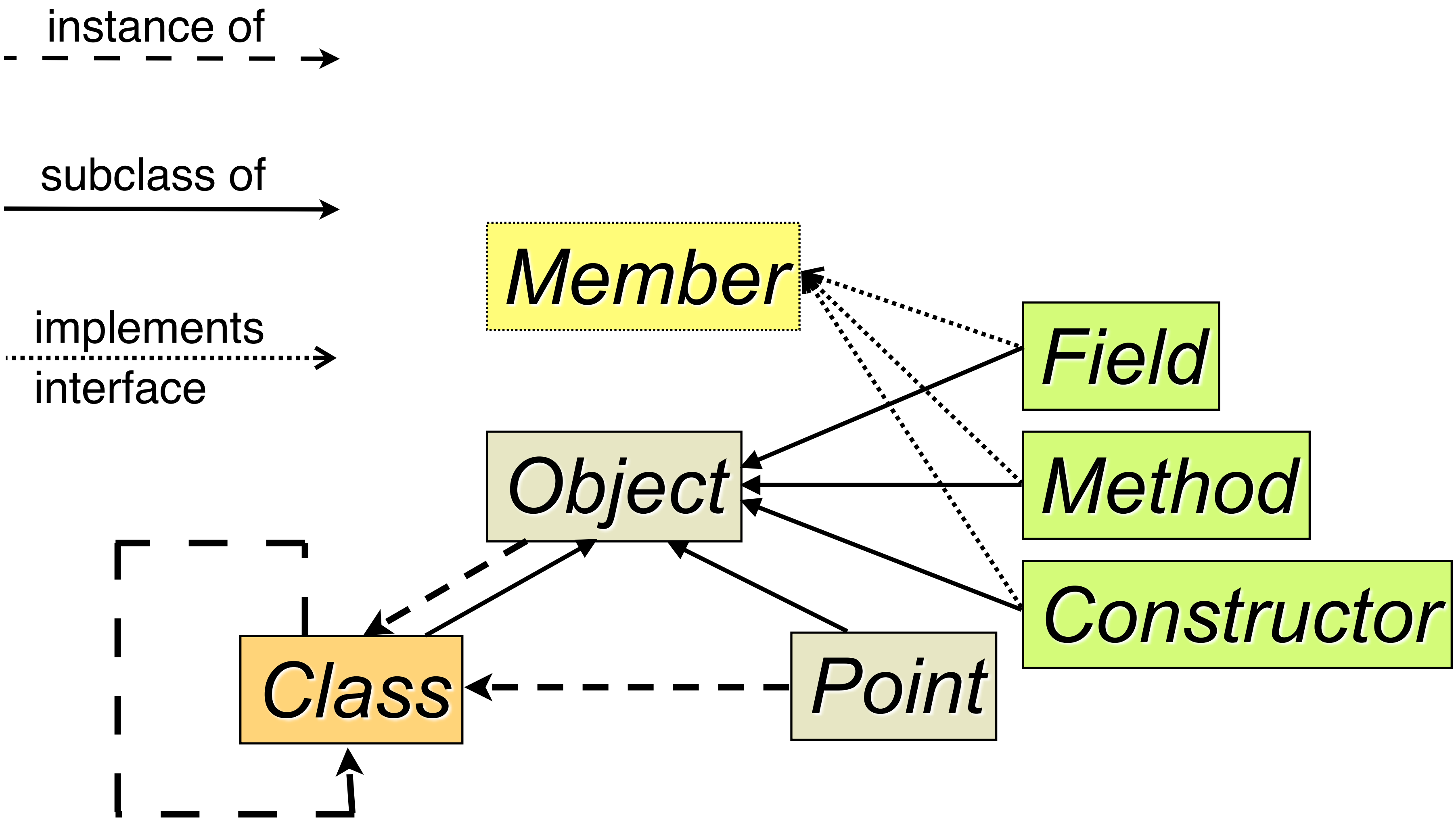
Objects are instances of a class

Classes are instances of the meta class Class

The meta class Class is an instance of itself



The Java Meta-Object Protocol



10 a.3 The Java Reflection API

- Class
- Field, Method and Constructor

The Java Core Reflection API

provides a small, type-safe, and secure API

that supports introspection about classes and objects in the current JVM

If permitted by security policy, the API can be used to:

- construct new class instances and new arrays
- access and modify fields of objects and classes
- invoke methods on objects and classes
- access and modify elements of arrays

Intercession on classes and objects is forbidden

java.lang.reflect is in a subpackage of java.lang

Defines the following classes and methods:

- The classes Field, Method and Constructor
  - reify class and interface members and constructors
  - provide reflective information about the underlying member or constructor
  - a type-safe means to use the member or constructor to operate on Java objects
- Methods of class Class that enable the construction of new instances of the Field, Method and Constructor classes.
- and more...

There are also parts of the java.lang package that support reflection

- In particular the class Class

Class

Instances of the class Class represent classes *and interfaces* in a running Java application.

- Every array also belongs to a class that is reflected as a Class object
  - shared by all arrays with same element type and dimension
- Primitive Java types (**boolean**, **byte**, **char**, **int**, ...), and void are also represented as **Class** objects

Class has no public constructor

- **Class** objects are constructed automatically by the Java Virtual Machine as classes are loaded
- and by calls to **defineClass** method in the class loader

Defined in **java.lang** (instead of **java.lang.reflect**)

Using `Class`: Examples

Using a **Class** object to print the class name of an object:

```
void printClassName(Object obj) {
  System.out.println("The class of " + obj + " is "
                     + obj.getClass().getName());
 }
```

Getting the Class object for a named type using a class literal:

```
System.out.println("The name of class Foo is:"
                   + Foo.class.getName());
```

`Class`: Querying for Methods

Method	Description
Method **getMethod**(String name, Class[] parameterTypes)	Returns a `Method` object that represents a public method (either declared or inherited) of the target `Class` object with 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)	Returns a `Method` object that represents a declared method of the target `Class` object with the signature specified by the second parameters
Method[] **getDeclaredMethods**()	Returns an array of `Method` objects that represent all of the methods declared by the target `Class` object

\+ **getConstructor**, **getField**, ...

`Class`: Dealing with Type Representation

Method	Description
String **getName**()	Returns the fully qualified name of the target `Class` object
Class **getComponentType**()	If the target object is a `Class` object for an array, returns the `Class` object representing the component type
boolean **isArray**()	Returns `true` if and only if the target `Class` object represents an array
boolean **isInterface**()	Returns `true` if and only if the target `Class` object represents an interface
boolean **isPrimitive**()	Returns `true` if and only if the target `Class` object represents a primitive type or `void`

Field, Method and Constructor

- Three classes to reason about Java members
- Only JVM may create instances of these classes
  - i.e., they are final
- Can be used to manipulate the underlying objects
  - get reflective information about the underlying member
  - get and set field values
  - invoke methods on objects or classes
  - create new instances of classes
- These classes all implement the Member interface
  - defines methods to query member for basic information:
    - the class implementing a member
    - the Java language modifiers for the member

Field

A Field object represents a reified field

- may be a class variable (a static field)
- or an instance variable (a non-static field)

Methods of class Field are used to

- obtain the type of the field
- get and set the field’s value on objects

Field

Method	Description
getDeclaringClass()	Returns the `Class` object representing the class or interface that declares the field represented by this `Field` object.
getModifiers()	Returns the Java language modifiers for the field represented by this `Field` object, as an integer.
getName()	Returns the name of the field represented by this `Field` object, as `String`.
getType()	Returns a `Class` object that identifies the declared type for the field represented by this `Field` object.
get(Object obj)	Returns the value of the field represented by this `Field` object, on the specified object. The value is automatically wrapped in an object if it has a primitive type.
set(Object obj, Object value)	Sets the field represented by this `Field` object on the specified object argument to the specified new value. The new value is automatically unwrapped if the underlying field has a primitive type.
toString()	Returns a `String` describing this `Field`.

Constructor

A **Constructor** object represents a reified constructor

Methods of class **Constructor** are used to

- obtain the formal parameter types of the constructor
- get the checked exception types of the constructor
- create and initialise new instances of the class that declares the constructor
  - provided the class is instantiable
  - using the method **newInstance**

Constructor

Method	Description
getDeclaringClass()	Returns the Class object representing the class that declares the constructor represented by this Constructor object.
getExceptionTypes()	Returns an array of Class objects that represent the types of of exceptions declared to be thrown by the underlying constructor represented by this Constructor object.
getModifiers()	Returns the Java language modifiers for the constructor represented by this Constructor object, as an integer.
getName()	Returns the name of this constructor, as a string.
getParameterTypes()	Returns an array of Class objects that represent the formal parameter types, in declaration order, of the constructor represented by this Constructor object.
newInstance(Object[] initargs)	Uses the constructor represented by this Constructor object to create and initialize a new instance of the constructor's declaring class, with the specified initialization parameters.
toString()	Returns a String describing this Constructor.

Method

A **Method** object represents a reified method

- may be an abstract method, an instance method or a class (**static**) method

Methods of class **Method** are used to

- obtain the formal parameter types of the method
- obtain its return type
- get the checked exception types of the method
- invoke the method on target objects
  - instance and abstract method invocation uses dynamic method resolution
    - based on target object's run-time class
  - static method invocation uses the static method of the method's declaring class

Method

Method	Description
Class **getDeclaringClass**()	Returns the `Class` object that declared the method represented by this `Method` object
Class[] **getExceptionTypes**()	Returns an array of `Class` objects representing the types of the exceptions declared to be thrown by the method represented by this `Method` object
int **getModifiers**()	Returns the modifiers for the method represented by this `Method` object encoded as an `int`
String **getName**()	Returns the name of the method represented by this `Method` object
Class[] **getParameterTypes**()	Returns an array of `Class` objects representing the formal parameters in the order in which they were declared
Class **getReturnType**()	Returns the `Class` object representing the type returned by the method represented by this `Method` object
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

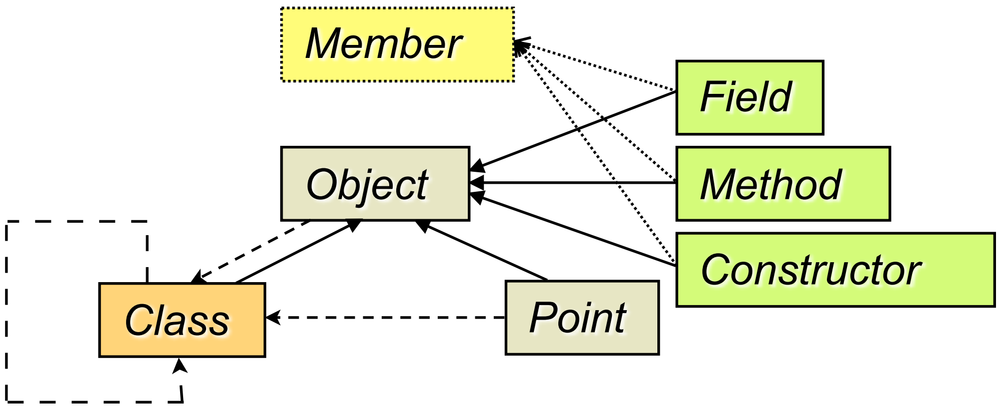
10 a.4

History of Reflection in Java

History of Reflection in Java

Since Java ≤1.2

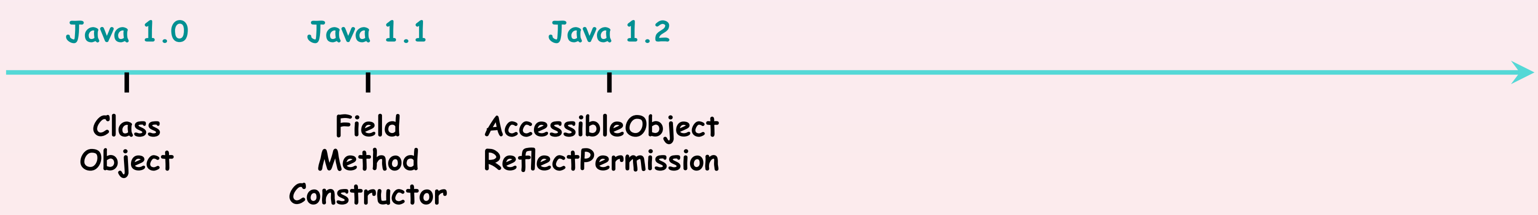
- java.lang.Object
  - java.lang.Class
  - java.lang.reflect.Member
    - java.lang.reflect.Field (Member)
    - java.lang.reflect.Method (Member)
    - java.lang.reflect.Constructor (Member)



History of Reflection in Java

Since Java ≤1.2

- `java.lang.Object`
  - `java.lang.Class`
  - `java.lang.reflect.Member`
    - `java.lang.reflect.Field` (Member)
    - `java.lang.reflect.Method` (Member)
    - `java.lang.reflect.Constructor` (Member)
- `boolean.class`, `char.class`, `int.class`, `double.class`, ...



Reflective Features Since 1.2

AccessibleObject

- base class for Field, Method and Constructor objects
- setting the accessible flag in a reflected object suppresses default Java language access control checks when it is used
  - permits sophisticated applications with sufficient privilege, such as Java Object Serialization or other persistence mechanisms, to manipulate reflected objects in a manner that would normally be prohibited

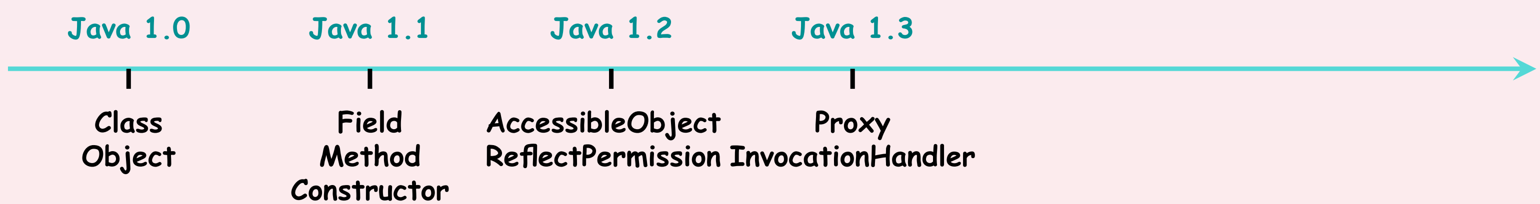
ReflectPermission

- is the security permission class for reflective operations
- defines the suppressAccessChecks permission name which allows suppressing the standard Java language access checks (for public, default (package) access, protected, and private members) performed by reflected objects at their point of use

History of Reflection in Java

Since Java 1.3

- `java.lang.Object`
  - `java.lang.Class`
  - `java.lang.reflect.Member`
  - `java.lang.reflect.AccessibleObject`
    - `java.lang.reflect.Field` (Member)
    - `java.lang.reflect.Method` (Member)
    - `java.lang.reflect.Constructor` (Member)
  - `java.lang.reflect.Proxy`
  - `java.lang.reflect.InvocationHandler`
- `boolean.class`, `char.class`, `int.class`, `double.class`, ...



Reflective Features Since 1.3

Proxy (class)

- provides static methods for creating dynamic proxy classes and instances
- is superclass of all dynamic proxy classes created by those methods

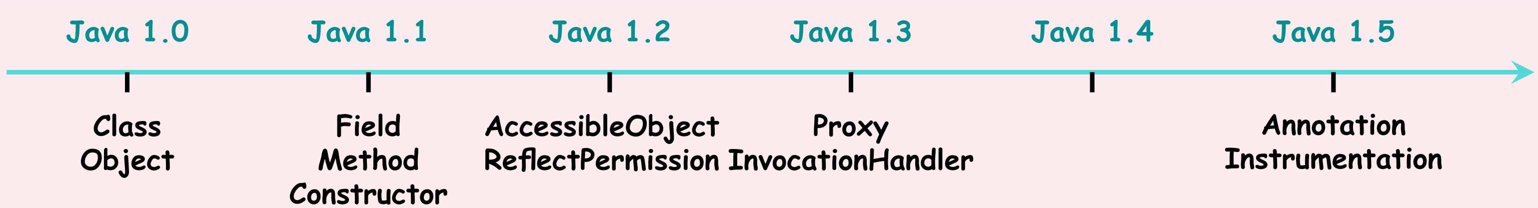
InvocationHandler (interface)

- is the interface implemented by the invocation handler of a proxy instance
- each proxy instance has an associated invocation handler
- when a method is invoked on a proxy instance, the method invocation is encoded and dispatched to the invoke method of its invocation handler

History of Reflection in Java

Since Java 1.5

- java.lang.Object
  - java.lang.Class
  - java.lang.reflect.Member
  - java.lang.reflect.AccessibleObject
    - java.lang.reflect.Field (Member)
    - java.lang.reflect.Method (Member)
    - java.lang.reflect.Constructor (Member)
  - java.lang.reflect.Proxy
  - java.lang.reflect.InvocationHandler
  - java.lang.annotation.Annotation
  - java.lang.instrument.Instrumentation
- boolean.class, char.class, int.class, double.class, ...



Reflective Features Since 1.5

Annotation

Java 1.5 supports annotating Java programs with custom annotations

Annotations can be accessed at compile-time **and at run-time**

E.g., annotate some methods with *@prelog* annotation and check for this annotation to print a log message before execution of those methods

Instrumentation

java.lang.instrument provides services that allow Java programming agents to instrument programs running on the JVM

The instrumentation mechanism is **modification** of the byte-code of methods.

Type

reflection API of Java 1.5 was extended to deal with new Java 1.5 types

- generic arrays, parameterized types, type variables, ...

Type is the common superinterface for all Java types

- several subinterfaces for the new Java 1.5 types

Lecture 10 b

Advanced Reflection in Java

Partly inspired on :

- Chapter 5 of Book “Java Reflection in Action”
- Slides by Prof. Walter Cazzola

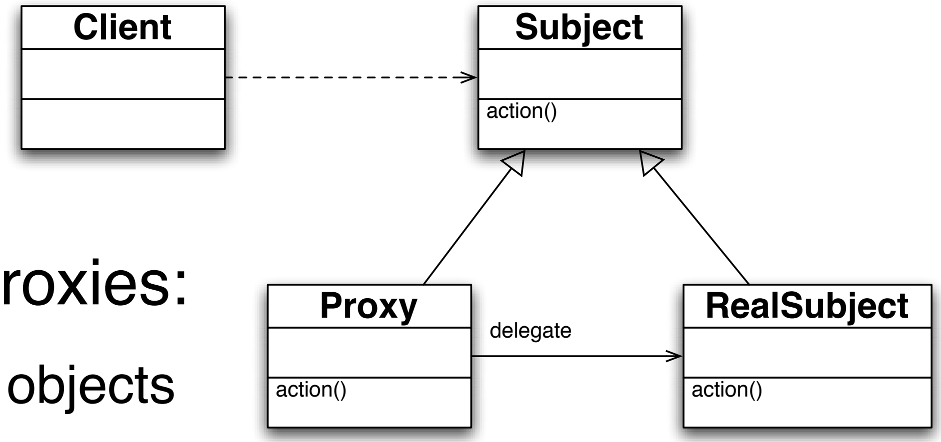
10 b.1 Dynamic Proxies

- Proxy
- InvocationHandler

Proxy Design Pattern

The **proxy pattern** defines a proxy as a surrogate for another object to control access to it

- a proxy keeps a reference to the real object,
- is substitutable with it (identical interface)
- and delegates requests to it



Typical examples of usage of proxies:

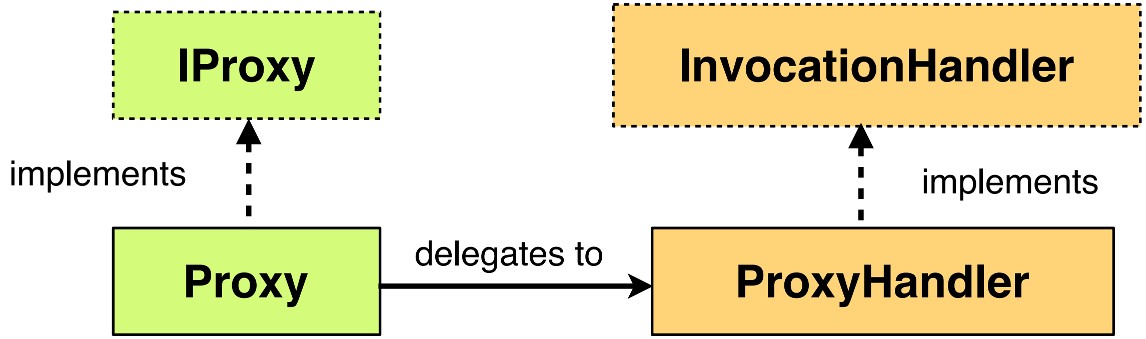
- local representation of remote objects
- access protection for secure objects
- delay of expensive operations
- …

Dynamic Proxies

To easily implement proxies, Java 1.3 introduced the Dynamic Proxy API

A dynamic proxy requires

- an instance of the Proxy class
- a proxy interface
  - this is the interface that is implemented by the proxy class
  - interestingly, one proxy can implement multiple interfaces
- each proxy instance has an InvocationHandler

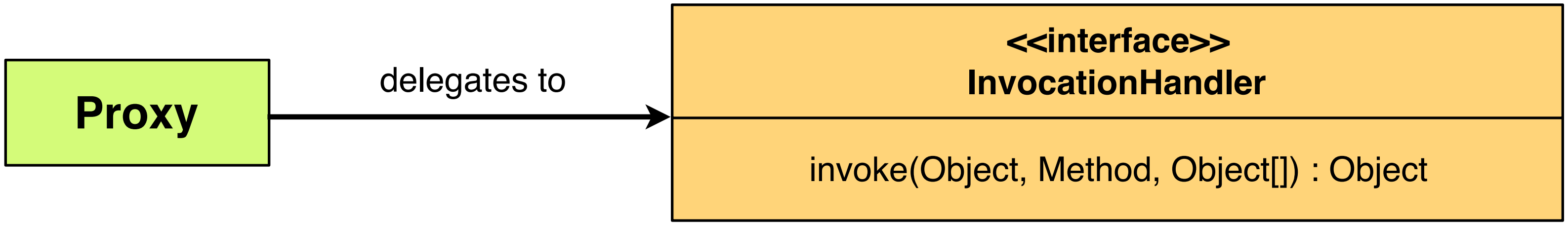


InvocationHandler

Each proxy instance has an `InvocationHandler`

- The invocation handler determines how to treat messages that are sent to the proxy instance
- When a method is invoked on a proxy instance, the method invocation is encoded and dispatched to the `invoke()` method of its invocation handler

```
Object invoke(Object proxy, Method m, Object[] args)
```



Example of an InvocationHandler


```
public class TraceHandler implements InvocationHandler {

    // the real object the proxy delegates to
    private Object baseObject;

    public TraceHandler(Object base) {
        baseObject = base;
    }

    public Object invoke(Object proxy, Method m, Object[] args) {
        Object result = null;  // result to be returned by the method
        try {
            System.out.println("before " + m.getName());
            result = m.invoke(baseObject, args);
            System.out.println("after " + m.getName());
        }
        catch (Exception e) {
            e.printStackTrace();
        }
        return result;
    }
}
```

An InvocationHandler to Trace method calls

Proxy Class

The class `Proxy` provides static methods for creating dynamic proxy classes and instances

Is also the superclass of all dynamic proxy classes created by those methods.

To create a proxy for some class `Foo` :

```
InvocationHandler handler = new MyInvocationHandler(...);
Class proxyClass = Proxy.getProxyClass(
  Foo.class.getClassLoader(), new Class[] { Foo.class });
Foo f = (Foo) proxyClass
  .getConstructor(new Class[] { InvocationHandler.class })
  .newInstance(new Object[] { handler });
```

or more simply:

```
Foo f = (Foo) Proxy.newProxyInstance(
       Foo.class.getClassLoader(),
      new Class[] { Foo.class }, handler);
```

Creating a new proxy instance for some class

Example: Proxy That Traces Method Calls


```
public class Component1 implements IComponent {

  Color myColor;

  public Color getColor() {
    System.out.println("Inside the getColor() method of
Component1.");
    return myColor;
  }

  public void setColor(Color color) {
    myColor = color;
    System.out.println("The color of component 1 has
been set.");
  }

}
```

real object to which proxy will delegate

Creating a proxy to trace method calls

```
public interface IComponent {

   public Color getColor();

   public void setColor(Color color);

}
```

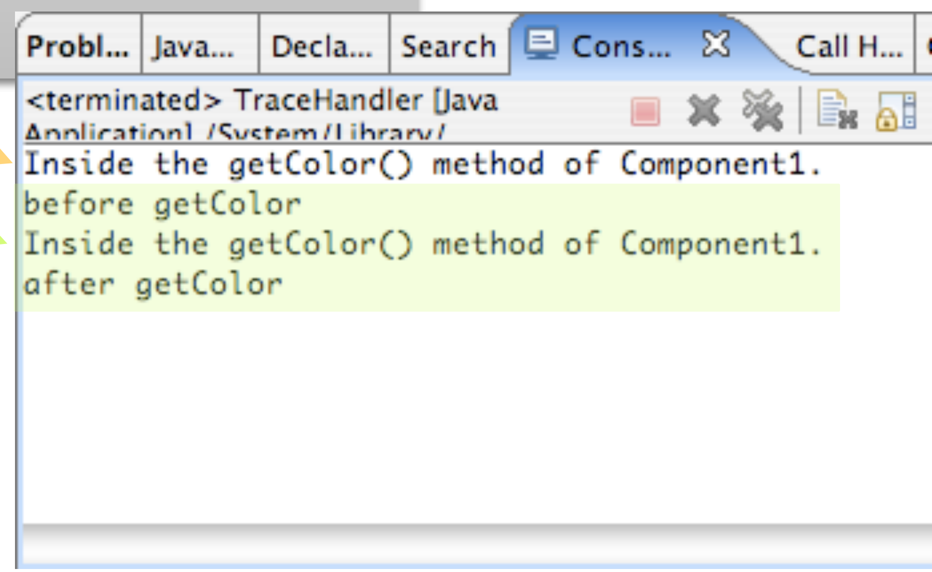
interface shared by real object and proxy object

Example: Proxy That Traces Method Calls

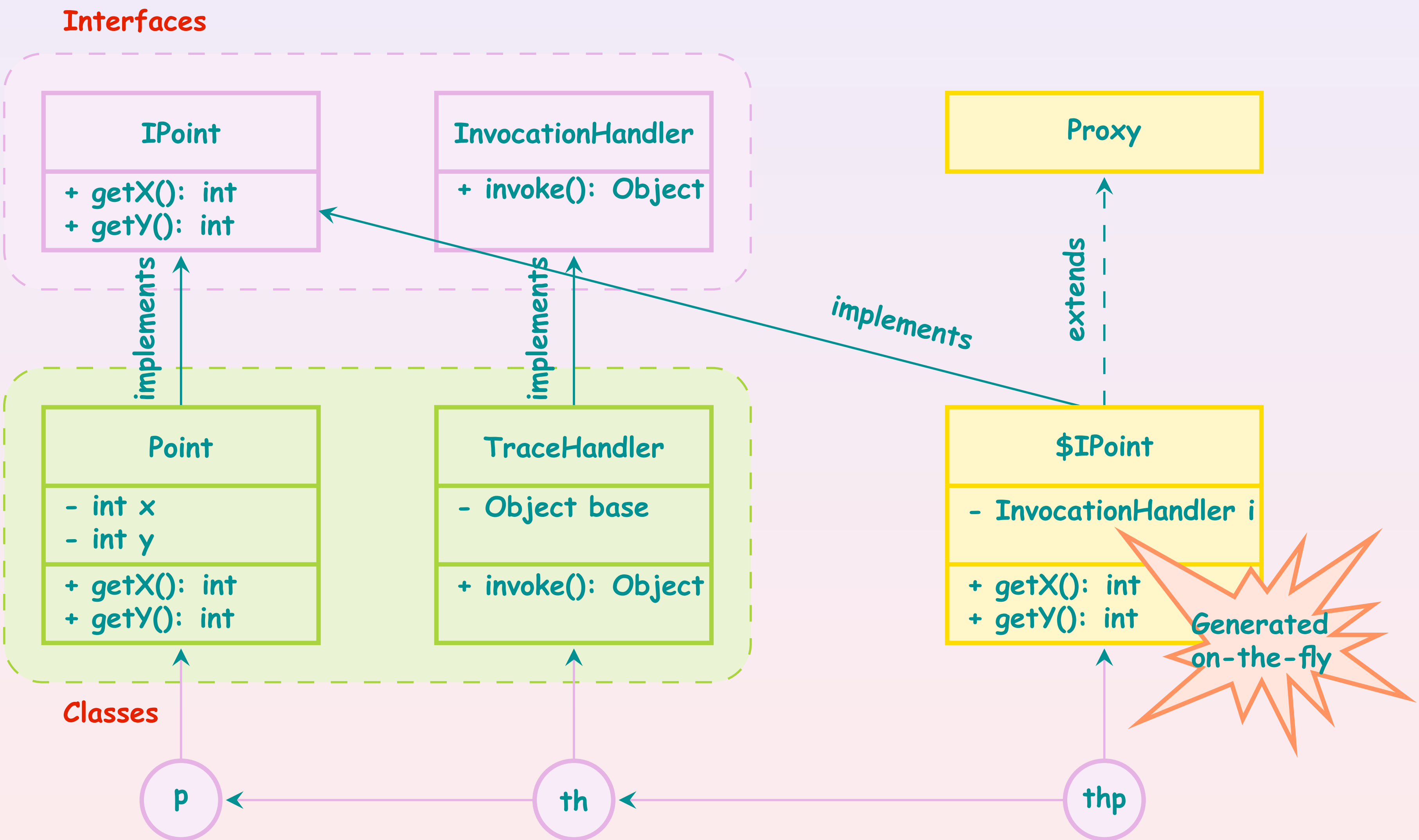
```java
IComponent c1 = new Component1(new Color(0));

InvocationHandler th = new TraceHandler(c1);
IComponent proxy = (IComponent) Proxy.newProxyInstance(
  c1.getClass().getClassLoader(),
  c1.getClass().getInterfaces(),
  th);
/* standard call */
c1.getColor();
/* traced call */
proxy.getColor();
```

creating the proxy instance

Creating a proxy to trace method calls



How Java Creates A Proxy



10 b.2 Call Stack Introspection

- Throwable
- StackTraceElement

Call Stack Introspection

- State introspection
- Call stack
- Reifying the call stack
  - Throwable
  - StackTraceElement
- Examples:
  - printing the call stack
  - show warnings for unimplemented methods

State Introspection

- Introspection is not only application structure introspection
- Some information about the program execution can be introspected as well
  - the execution state
  - the call stack
- Each thread has a call stack consisting of stack frames
- Call stack introspection allows a thread to examine its context
  - the execution trace and the current frame

Call Stack

```java
1. package reflectionexample;
2.
3. public class Example {
4.
5.     public static void m1() {
6.         m2();
7.     }
8.
9.     public static void m2() {
10.        m3();
11.    }
12.
13.    public static void m3() {
14.        // do something
15.    }
16.
17.    public static void main(String[] args) {
18.        m1();
19.    }
20.
21. }
```

```
Call Stack:(m3)

class: Example
method: m3
line: 14

class: Example
method: m2
line: 10

class: Example
method: m1
line: 6

class: Example
method: main
line: 18
```

Reifying the Call Stack

In Java there is **no** accessible CallStack meta-object

Throwable

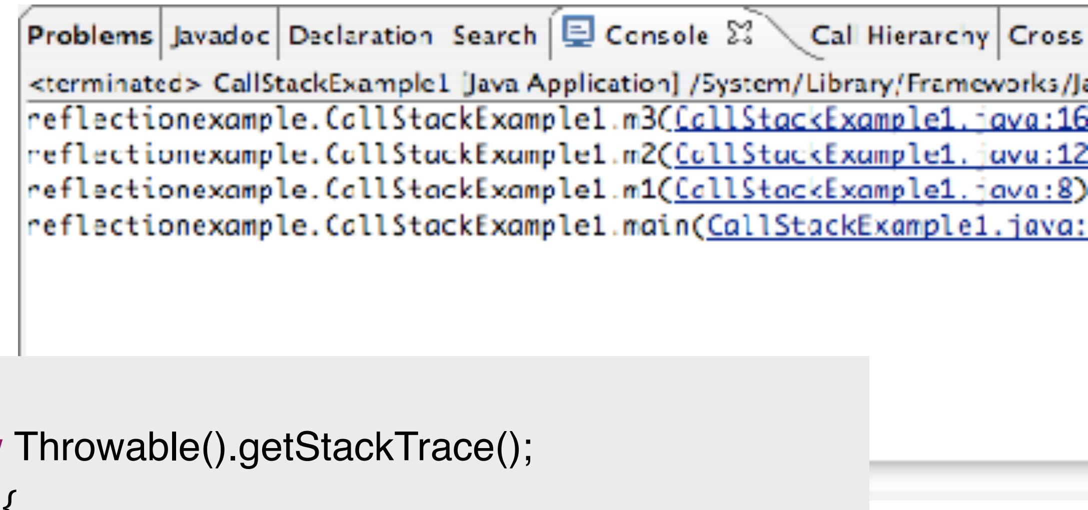
But...

- When an instance of **Throwable** is created, the call stack is saved as an array of **StackTraceElement**
- By writing `new Throwable().getStackTrace()`
- we gain access to a *representation* of the call stack at the moment when Throwable was created.
- The getStackTrace() method returns the current call stack as an array of StackTraceElement.
  - The first frame (position 0) is the current frame

Only introspection !

Example: Printing the Call Stack


```java
public class CallStackExample1 {

    public static void m1() {
        m2();
    }

    public static void m2() {
        m3();
    }

    public static void m3() {
        StackTraceElement[] stack = new Throwable().getStackTrace();
        for (int i = 0; i < stack.length; i++) {
            System.out.println(stack[i]);
        }
    }

    public static void main(String[] args) {
        m1();
    }
```



Example: Warnings For Unimplemented Methods

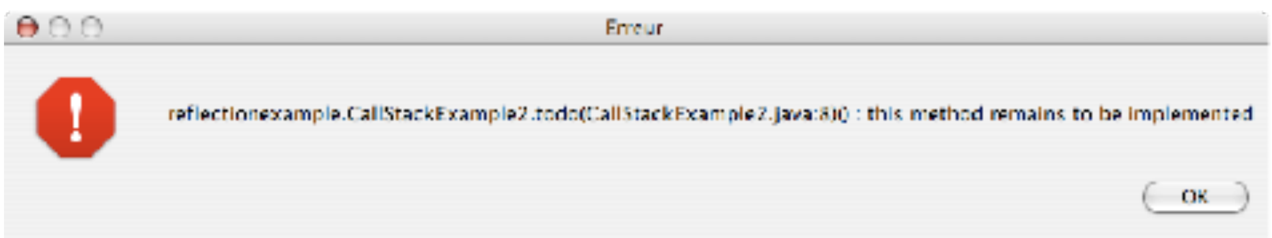
How to create an auxiliary method to be used as placeholder for unimplemented methods?

I have a method **todo()** remaining to be implemented

I provide the following dummy implementation

```
public static void todo() {
    toBeImplemented();
}
```

When calling that method **todo()** I want to get a warning:



as well as a warning message on the Console:

Warning: reflectionexample.CallStackExample2.todo(CallStackExample2.java:8)() : this method remains to be implemented

Example: Warnings For Unimplemented Methods

```
public class CallStackExample2 {

    public static void todo() {
        toBeImplemented();  }

    public static void toBeImplemented() {
        // Get the stack element referring to the method calling this one
        StackTraceElement el = new Throwable().getStackTrace()[1];
        // Show a dialog box with information on the method remaining to be implemented
        String msg = el.toString() + "() : this method remains to be implemented”;
        // Show this message in a dialog box
        JOptionPane.showMessageDialog(null, msg, "Erreur", JOptionPane.ERROR_MESSAGE);
        // Print this warning on the console
        System.err.println("Warning: " + msg);  }

    public static void main(String[] args) {
        todo();  }

}
```

Reifying The Call Stack

StackTraceElement:

class: Example
method: m3
line: 14
filename: Example.java

“myPackage.Example.m3(Example.java:14)”

StackTraceElement

From a frame we can get:

getFileName()	the filename containing the execution point
getLineNumber()	the line number where the call occurs
getClassName()	the name of the class containing the execution point
getMethodName()	the name of the method containing the execution point

10 b.3 Instrumentation

java.lang.instrument

Suppose We Want To…

1. Instrument a Java program to print a message on the console whenever a class is loaded by the JVM
2. Instrument a Java program to print all messages being sent while the program is running
3. Replace the definition of a class at runtime
   - even for classes with currently active instances
   - for example, to change the behaviour of some messages understood by those instances

... then Java instrumentation may be your answer

Goal of Java Instrumentation

java.lang.instrument

Provides services that allow Java programming language agents to instrument programs running on the JVM.

This instrumentation is achieved by modifying bytecode.

Allows you to create agents that run embedded in a JVM and *intercept the classloading process*, to

- monitor the classloading process
- modify the bytecode of classes

Can be combined with dedicated libraries for Java bytecode manipulation, such as **Javassist** and **BCEL**.

Mechanics of Java Instrumentation

To implement an agent that intercepts class loading you need to define:

A class implementing the premain method:

```
public static void premain(String agentArguments,
                           Instrumentation instrumentation) { ... }
```

A class implementing a transformer (which describes how the bytecode should be transformed):

```
public class SomeTransformer implements ClassFileTransformer
public byte[] transform(ClassLoader loader,
                        String className, Class redefiningClass,
                        ProtectionDomain domain, byte[] bytes)
                        throws IllegalClassFormatException { ... }
```

You can also put both methods in a single class.

Mechanics of Java Instrumentation

The transform method receives information on the class to be loaded and can modify its bytecode.

```
public byte[] transform(ClassLoader loader,
                    String className, Class redefiningClass,
                    ProtectionDomain domain, byte[] bytes)
```

- returns null if there's no modification, else returns the new byte code
- to modify the bytecode you can use a specialised library like Javassist

The premain method should add the transformer to the agent:

```
public static void premain(String agentArguments,
                          Instrumentation instrumentation) {
  instrumentation.addTransformer(new SomeTransformer());
}
```

Mechanics of Java Instrumentation

To plug the agent into the JVM and execute it, you need to put it inside a .jar file:

```
jar cmf ManifestFile.txt MyAgent.jar
        MyAgent.class SomeTransformer.class
```

The manifest file for this jar has to declare the premain class:

```
Premain-Class: MyAgent
```

If the agent modifies the bytecode, this also has to be declared in the manifest file:

```
Can-Redefine-Classes: true
```

Finally, to instrument a Java program you need to add the javaagent parameter when you execute the JVM:

```
> java -javaagent:MyAgent.jar MyProgram
```

Examples

1. Instrument a Java program to print a message on the console whenever a class is loaded by the JVM

2. Instrument a Java program to print all messages being sent while the program is running

3. Replace the definition of a class at run time
   - even for classes with currently active instances
   - for example, to change the behaviour of some messages understood by those instances

Example 1: Instrument Java Code


```
/**
 * Just prints a message to stdout before each Class is loaded
 * > java -javaagent:LogClassLoad.jar <the_Main_Class_to_execute>
 */
public class LogClassLoad implements ClassFileTransformer {

    public static void premain(String options, Instrumentation ins) {
        ins.addTransformer(new LogClassLoad());
    }

    public byte[] transform(ClassLoader loader, String className, Class cBR,
                    java.security.ProtectionDomain pD,
                    byte[] classfileBuffer)
            throws IllegalClassFormatException {
        try {
            System.out.println("LOADING: " + className);
        }
        catch (Throwable exc) {
                System.err.println(exc);
        }
        return null; // For this first example no transformation is required:
                // we are only logging when classes are loaded
    }
}
```

Example 1: Instrument Java Code


```
➜ java -javaagent:LogClassLoad.jar example/Application                    …
LOADING: java/util/function/Function
…
LOADING: java/lang/ClassValue
…
LOADING: java/lang/invoke/MethodHandleStatics
LOADING: java/lang/invoke/MethodHandleStatics$1
…
LOADING: java/lang/Package
LOADING: example/Application
LOADING: sun/launcher/LauncherHelper$FXHelper
LOADING: java/lang/Class$MethodArray
LOADING: java/lang/Void
LOADING: example/di/Container
…
LOADING: example/acme/GreatInventory
LOADING: example/model/Inventory
…
LOADING: example/acme/CoolLibrary
LOADING: example/model/Library
…
```

Examples

1. Instrument a Java program to print a message on the console whenever a class is loaded by the JVM
2. Instrument a Java program to print all messages being sent while the program is running
3. Replace the definition of a class at run time
   - even for classes with currently active instances
   - for example, to change the behaviour of some messages understood by those instances

Example 2: Print All Messages


```
public class TraceMethodCall implements ClassFileTransformer {
  public static void premain(String options, Instrumentation ins) {
    ins.addTransformer(new TraceMethodCall());
  }
  public byte[] transform(ClassLoader loader, String className, Class cBR,
                          ProtectionDomain pD, byte[] classfileBuffer)
                         throws IllegalClassFormatException {
    try {
      if(isSystemClass(className)) return null;

      ClassPool pool = ClassPool.getDefault();
      CtClass cc = pool.get(className);
      CtMethod[] methods = cc.getDeclaredMethods();

      for(CtMethod method : methods) {
        try {
          method.insertBefore("TraceMethodCall.trace();");
        } catch (Exception e) {}
      }

      return cc.toBytecode();
    }
    catch (Throwable exc) { … }
  }
```

```
import javassist.ClassPool;
import javassist.CtClass;
import javassist.CtMethod;
```

Example 2: Print All Messages


```
public class TraceMethodCall implements ClassFileTransformer {
   …
   /** Check if a class is a system class, based on the package name. **/
   public static boolean isSystemClass(String className) { … }

   public static boolean isDottedSystemClass(String className) { … }
   …
   public static void trace() {
     Throwable thw = new Throwable();

     if(thw.getStackTrace().length > 2) {
          StackTraceElement stackTo = thw.getStackTrace()[1];
          StackTraceElement stackFrom = thw.getStackTrace()[2];
          if(!isDottedSystemClass(stackFrom.getClassName())) {
               System.out.println("\"" + stackFrom.getClassName() + "."
                    + stackFrom.getMethodName() + "\" -> " + "\""
                    + stackTo.getClassName() + "." + stackTo.getMethodName() + "\"");
               }
          }
     }

}
```

Example 2: Print All Messages


```java
public class CallStackExample0 {

    public static void m1() { m2(); }

    public static void m2() { m3(); }

    public static void m3() { }

    public static void main(String[] args) { m1(); }

}
```

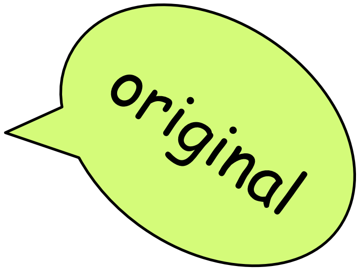
```
➜ java -javaagent:TraceMethodCall.jar -cp javassist.jar:. CallStackExample0

Transforming CallStackExample0
… method m1
… method m2
… method m3
… method main

"CallStackExample0.main" -> "CallStackExample0.m1"
"CallStackExample0.m1" -> "CallStackExample0.m2"
"CallStackExample0.m2" -> "CallStackExample0.m3"
```

Examples

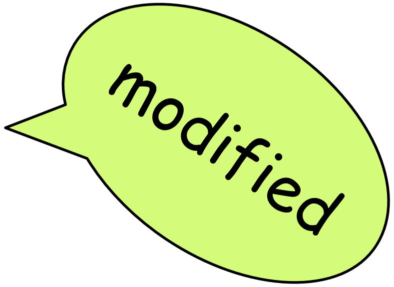
1. Instrument a Java program to print a message on the console whenever a class is loaded by the JVM
2. Instrument a Java program to print all messages being sent while the program is running
3. Replace the definition of a class at run time
   - even for classes with currently active instances
   - for example, to change the behaviour of some messages understood by those instances

Example 2: Replacing a Class (Goal)


```
/**
 * This is the original class that will be replaced by a modified copy at run
time.
 */
public class SwappedClass {
    public void message() {
        System.out.println("I am the original SwapedClass");
    }
}
```




```
/**
 * This modified class replaces the original class at run time.
 * The modified class has the same name but is located in a
subdirectory.
 * It is a clone of the original class where one method was changed.
 * Instances of this class will react differently to those messages.
 */
public class SwappedClass {
    public void message() {
        System.out.println("I am the MODIFIED SwapedClass");
    }
}
```



Example 2: Replacing a Class (Mechanics)


```
import java.lang.instrument.Instrumentation;

public class SwapClass {
    private static Instrumentation instCopy;

    public static void premain(String options, Instrumentation inst) {
        instCopy = inst;
    }

    public static Instrumentation getInstrumentation() {
        return instCopy;
    }

}
```

to dynamically replace a class, do:

```
SwapClass.getInstrumentation().redefineClasses(...)
```

Example 2: Replacing a Class (Code)


```
public class SwapTest {
   public static void main (String[] args) {
      try {
         // Create an instance of the class that will be modified
         SwappedClass swapped = new SwappedClass();
         // Send a message to the instance; the original message should be displayed
         swapped.message();
         // Now replace the class by a modified version
         // 1. read the class definition from file
         FileChannel fc = new FileInputStream(
                            new File("modif/SwappedClass.class")).getChannel();
         ByteBuffer buf = fc.map(FileChannel.MapMode.READ_ONLY, 0, (int)fc.size());
         byte[] classBuffer  = new byte[buf.capacity()];
         buf.get(classBuffer, 0, classBuffer.length);
         // 2. use instrumentation API to replace the old class’s definition by the new one
         SwapClass.getInstrumentation().redefineClasses(
            new ClassDefinition[] {
               new ClassDefinition(swapped.getClass(), classBuffer)});
         // Send a message to the old instance; the MODIFIED message will be displayed
         swapped.message();
      }
      catch (Exception e){
         System.out.println(e);
      }
   }
}
```

Example 2: Replacing a Class (At Work)


```
➜ java -javaagent:SwapClass.jar SwapTest
I am the original SwappedClass
I am the MODIFIED SwapedClass
```

Lecture 10

Reflection in Java : Conclusions

Reflection in Java: Conclusions

Benefits

- reflection in Java opens up the structure and the execution trace of the program;
- the reflection API is (relatively) simple and quite complete

Drawbacks

- reflection in Java is (mostly) limited to introspection;
- there isn't a clear separation between base and meta-level;
- reflection in Java can be inefficient

  … though there is progress is this regard

Possible Questions

- Explain the difference between introspection and intercession. Which of these is mostly supported by Java's reflection API?
- Discuss (with a concrete example) how reflective programming could be used to make a program more adaptable.
- What are the possible problems of reflection with respect to maintainability?
- In Java's meta-object protocol, what does the class Object represent? What happens when you ask this class for its class? What Java instruction can you use to retrieve that class?
- In Java's meta-object protocol, what does the class Class represent? What happens when you ask this class for its class? What Java instruction do you need to use to do that?
- How is the call stack reified in the Java reflection API? What kind of reflection (intercession or introspection) can be used to manipulate this call stack. What are the consequences?