OOP
Reflection

Kasper Østerbye
Mette Jaquet
Carsten Schuermann
IT University Copenhagen

1

Today's schedule

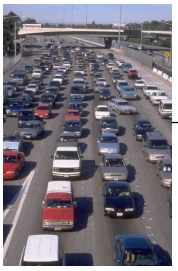
- Reflection
  - modelling a domain vs. modelling objects
- Testing
  - Example: Checking that all fields are private and have a getter method
  - A unit testing framework
- Object Irrelevance
- Annotations
  - Adding metadata to programs
- Summary

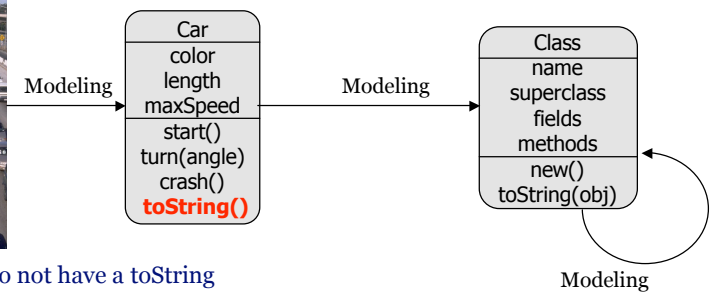
2

Philosophically aside

The objects in the application models aspects in the application domain.

E.g.





But real cars do not have a toString method.

•The solution is to model the classes too.

•A class whose instances are classes, is called a *metaclass*.

- The program contains not only a model of the application objects, but *also* a model of itself, which is why it is called reflective

From a symmetry argument, there has to be a model of the model of the model etc. This is often 'solved' by a loop.

3

Different object oriented programming languages handle the notion of meta classes very differently.

- Classes are not modelled in the language – C++ is an example of this
- Classes are modelled in the language, but cannot be changed at runtime (Java is an example of this).
- Classes are modelled in the language, and can be changed at runtime – Smalltalk, Clos, and some research versions of Java are examples.

There is also an interesting naming issue involved in the notion of Metaclasses.

Normally the name of a class is really a common name of its instances, not the class itself. Car is the name of a class whose instances are (models of) car's.

Therefore, the class named Class above is the class whose instances are classes.

In java, class Class is an instance of it self (Talk about a chicken and egg solution – the chicken is the egg!).

Fortunately, one does not have to worry too much on these philosophical issues to just use reflection for practical programming.

Concepts

A program which manipulates or examines itself is called *reflective*.

- *Introspection* means examining properties of the program
- *Intercession*/Manipulation, means changing properties of the program

- *Structural* reflection means to have access to the structure of the program, e.g. classes, methods, fields.
- *Behavioural* reflection means to have access to the run-time aspects, e.g. how objects are created, how methods are called.

- Reflection can take place at compile-time, load-time, or run-time.

- Traditionally one has in OOP only considered full reflection, that is both introspection and intercession, and both structural and behavioural, and at run-time.

- Nobody has ever succeeded in making full reflection efficient.

4

Usages of reflection

- To find out which methods a class has at run-time
  - Unit testing
  - Debugging
  - Plug and play systems
- Dynamic loading and maintenance of running systems
- Enterprise beans
- General storage and transmission of objects
- Redefining how method invocation is done, to achieve distribution.
- Transparent persistence.
- Programming language adaptation
- Object browsers
- Semantic checkers

In Java we can navigate the class structure using the java.lang.reflect package, and the class Class from java.lang.

Some of the methods of Class are:

- getClass() returns the class for any object.
- getDeclaredFields() returns an array of all fields in a class.
- getDeclaredMethods() returns an array of all methods in a class.
- newInstance() creates an instance of the class.
- getInterfaces() returns an array of interfaces

Some of the methods of Method are:

- getParameterTypes() returns an array of Class
- getModifiers() returns an integer representing which modifiers it has.
- invoke(Object, Object[]) calls the method.

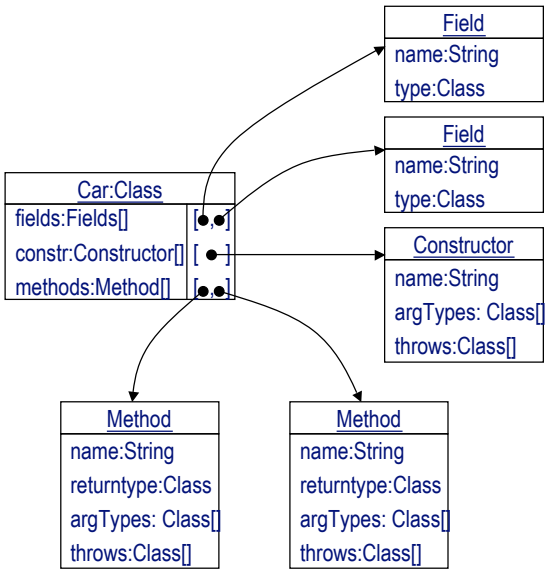
5

We will examine the practical usage of the reflection API in greater detail in the rest of the lecture.

It is an absolute necessity to examine and browse the API to solve the exercises. In the lecture we will examine some of the more important aspects, but not all.

The meta model – an example

```
class Car{
    private Person owner;
    private String make;
    public Car(String make, Person owner){
        this.make = make;
        this.owner = owner;
        owner.setCar(this);
    }
    public void setOwner(Person owner){
        this.owner.setCar(null);
        this.owner = owner;
        owner.setCar(this);
    }
    public String toString(){
        return make +" owned by " + owner;
    }
}
```



6

Please note that the reflective objects represent the *structure* of the program. The class Field represents the declaration of the field, not a field in a specific object.

Even if no instance of a Car is created the structure shown by the meta model above will exist.

Encapsulation checker

Assume we have a rule that all fields should be private, and they should have a public getter.

Write a program that gives an error for each field where this rule is broken.

The program to the right shows the outline of the program.

```
import java.lang.reflect.*;
class EncCheck {
    StringBuffer errorLog = new StringBuffer();
    Class clazz; // The class we are examining
    public void check(Class c)
        throws Exception {
        clazz = c; //… do the check and print out errors
    }
    public static void main(String[] args)
        throws Exception {
        String className = args[0];
        Class c = Class.forName(className);
        EncCheck checker = new EncCheck();
        checker.check(c);
    }
}
-----
dos:>java EncCheck dk.oop.itu.lecture6.MyClass
```

7

We have not used the arguments for the main methods often, but here is an example. To use this program, we will call it from a prompt

java EncCheck name-of-class-file-to-be-checked.

Notice, for reflection to be used, the program to be examined must be able to compile. That is, the class we want to check must be compiled by the javac compiler first.

The reflection package is very very full of methods which can throw exceptions. The code for this example simply solves this by having nearly all methods throw Exception.

The code above uses the static method forName in class Class, to find the class object for the corresponding string name. This is one of two typical situations. Either we, like here, have the name of a class (or method or field), and need to find the corresponding object. The other typical situation is that we have an array of classes (or more often methods or fields), and we need to search for a specific one, or we need to do something for all of them.

The definition of forName is:

```
public static Class forName(String className)
        throws ClassNotFoundException
```

Do not confuse the class name Class with the keyword class. Class above is the return type of the method 'forName'.

Finding all fields

The first thing we need to do is to find all fields in the class, and make sure they are private.

Does the rule apply to static fields as well? We will ignore static fields.

The underlined aspects are part of the reflection package, the rest is plain java.

```
List<Field> nonStaticFields;

public void findAllNonStaticFields(){
    Field[] fields = clazz.getDeclaredFields();
    nonStaticFields = new ArrayList<Field>();
    for (Field f : fields){
        int modifiers = f.getModifiers();
        if (! Modifier.isStatic(modifiers) )
            nonStaticFields.add(f);
    }
}
```

8

The method getDeclaredFields returns all the fields declared in the class, but it does not give us the fields of the super class. However, we can check the superclass independently, because private fields cannot be used in the subclass anyway.

There is also a method called getFields that returns all public fields that are declared in a class or inherited from its superclass. This method however does not return private and protected fields.

All methods in the reflection package that return collections, use plain arrays for this.

The for loop examines each field in turn.

The getModifiers method returns an integer, which might at first seem quite strange. The integer contains a coding of a set of modifiers. In the class Modifier, there is a range of methods which can be used to check if a given modifier is in that set.

The implementation is using the integer as a *bitmap*. If we look at only three modifiers public, static, final, we can say that public is represented by bit 0, static by bit 1, and final by bit 2. A public static final field is then represented as $2^0+2^1+2^2 = 7$, and one which is only final is represented as $2^2=4$. A public final is represented as $2^0 + 2^2 = 5$. This is a standard trick in programming, it is very efficient for small sets.

The modifiers do therefore only exist as bit-patterns in an integer, and it is not possible to make methods on integers. Therefore there is a class Modifier, which has all its methods static, and expect the argument to be a bitmap encoded the right way.

If you do not want to be bothered by bitmaps, just note the way it is used in the code above, that is the standard way.

Checking that all non-static fields are private

This is a combination of using an iterator, and the somewhat funny way to check for the modifiers.

Alternatively, the check could be written in one line:

```
Modifier.isPrivate( f.getModifiers() )
```

Some find that more succinct

```
public void ensurePrivate() {
    for ( Field f : nonStaticFields ){
        int modifiers = f.getModifiers();
        if (! Modifier.isPrivate(modifiers) )
            errorLog.append
                (f.toString() + " is not private\n");
    }
}
```

9

The errorLog depends on the field having a toString method. It will print the type of the field, and the name of the field and what class the field is located in.

Check that there are get accessors for each field

Here we use the method getMethod on Class, which finds a method based on the signature of the method.

The signature of a method is its name, and the type of the parameters.

We follow the principle of accessors having the same name as the field, but prefixed by ”get”, and the first letter of the field name as upper case.

E.g. if the field is named foo, the getter is named getFoo.

```
private static final Class[] noParameters = new Class[0];

public void ensureGetters(){
    for ( Field f : nonStaticFields ){
        try{
            clazz.getMethod(getter(f), noParameters);
        }catch(NoSuchMethodException ex){
            errorLog.append
                (f.toString() + " has no accessor\n");
        }
    }
}
private String getter (Field f){
    String name = f.getName();
    String first = name.substring(0,0).toUpperCase();
    String rest = name.substring(1,name.length());
    return "get"+ first + rest;
}
```

10

The method getMethod will throw an exception if there is no method with the requested signature. We catch this, and write a note to the error log.

If we run the program with itself as input, we get the following warnings:

```
java.lang.StringBuffer EncCheck.errorLog is not private
java.lang.Class EncCheck.clazz is not private
java.util.List EncCheck.nonStaticFields is not private
java.lang.StringBuffer EncCheck.errorLog has no accessor
java.lang.Class EncCheck.clazz has no accessor
java.util.List EncCheck.nonStaticFields has no accessor
```

If we examine the first line, we can see that the field is printed in the format

```
FieldType FieldClass.FieldName
```

java.lang.StringBuffer is the type of the field, EncCheck is the name of the class containing the field, and errorLog is the name of the field in that class.

Specifying signatures

The method getMethod() takes as argument the name of the method, and an array of classes, which specify what parameters the method expects.

Assume we have a method

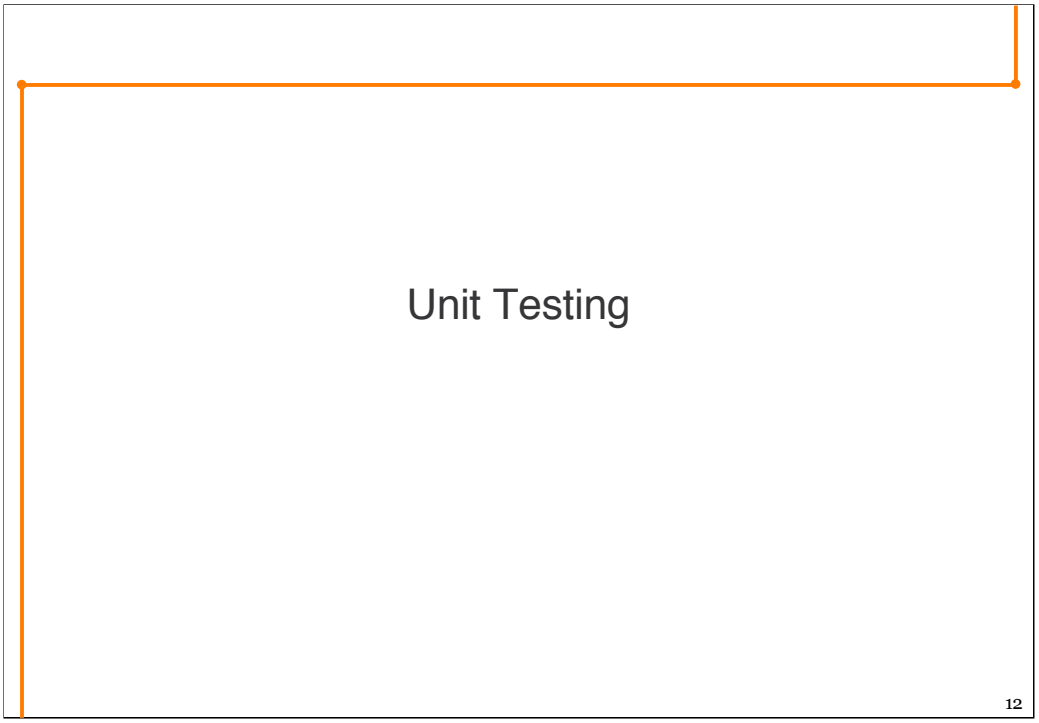
```
foo(boolean b , Person p, int[] i)
```

The parameters should be written up as the elements in an array
Such an array can be written as:

```
Class[ ] fooParam = new Class[ ] {boolean.class, Person.class, int[].class};
```

In general, one can obtain a class object for a given class, interface or primitive type as

```
type.class
```


Unit Testing
12

Test Driven Development

Unit testing is one of several core practices of the eXtreme Programming (XP) methodology.

Using a testing framework like JUnit, tests for all classes in a system can be collected in a test suite, and all tests can be run and evaluated by a push on a button.

Some benefits of unit testing:

- Quick feedback
- Automated validation of testresults
- Confidence in the code
- Collective code ownership
- Reuse of tests

13

XP features around 12 core practices and many of them are related and enhance the benefits of each other. Some of the other practices that work very well with test driven development are:

- Continuous integration
- Refactoring
- Simple Design
- Collective code ownership

Unit Testing

The principle is that together with the development of a class, one should also write a set of tests, to ensure that the class works as expected.

Unit testing step by step:

1. Write one test.
2. Compile the test. It should fail, as you haven't implemented anything yet.
3. Implement just enough to compile.
4. Run the test and see it fail.
5. Implement just enough to make the test pass.
6. Run the test and see it pass
7. Repeat from the top.

Test framework

A framework to be used for performing unit tests can be implemented using reflection.

The next slides are about the development of such a system and how to write the tests.

A typical test class might look as the class to the right.

Each method that is prefixed by ”test” should be executed. All failures are collected and printed in the main method.

```
public class MyTestClass extends TestClass
{   public void testParseInt(){
        check("'123' is 123",Integer.parseInt(”123”)==123 );
    }
    public void testParseDouble(){      //Should fail
        check("15 is 12.5",Double.parseDouble(”15”) == 12.5 );
    }
    public void testStringEquality(){
        String s = "abc";
        check("abc is aBc", s.equalsIgnoreCase("aBc"));
    }
}

// main method in some class – does not matter which
public static void main(String[] args){
    System.out.println(
    UnitTester.testTheClass("dk.itu.oop.lecture6.MyTestClass"));
}
```

15

The underlined method calls are those which play an important role in the test framework.

The essence of a TestClass is that it has a number of testMethods, each one containing one or more check’s.

When one runs a Test, each test method is executed, and the results of each test method is recorded, and printed in the end.

In the above, the testing framework has two visible parts. First, the class MyTestClass extends the general TestClass. The purpose is to inherit the check method used in each test method. Second, in the main method, the static method testTheClass gets the name of the class to be tested as argument.

The testTheClass method does the following steps:

1) Look up the class with the name given as parameter
2) Finds all methods that have a name starting with “test”
3) Execute those methods, collecting the results as we go along.

We will look at each in turn.

The TestClass

This is possibly the simplest TestClass which can be used.

It defines two check methods, each taking a boolean condition expected to be true. One also has a String parameter describing the test.

In addition a CheckFailure exception is declared. If a check fails, this exception is thrown.

```
public class TestClass {

    public class CheckFailure extends RuntimeException {
        public String msg;
        CheckFailure(String msg){
            this.msg = msg;
        }
    }

    protected void check(boolean expr) throws CheckFailure{
        if (!expr)
            throw new CheckFailure("Check failed");
    }
    protected void check(String msg, boolean expr)
                                throws CheckFailure{
        if (!expr)
            throw new CheckFailure(msg);
    }
}
```

16

The check methods are protected, as they are only intended to be used from subclasses of TestClass – for example in MyTestClass on the previous slide.

In the test framework JUnit, there are many different check methods, in particular many that compare two values

check(String msg, int expected, int observed)

check(String msg, double expected, double observed)

etc.

As part of the exercises, you are expected to extend the framework with some of these methods.

The unit tester

Note that the testTheClass method is static, but creates an instance of UnitTester in the body.

We will examine each of the three auxiliary methods in turn.

```
private void setClass(String className)
    throws ClassNotFoundException,
           InstantiationException,
           IllegalAccessException {
    testClass = Class.forName(className);
    testObject = testClass.newInstance();
}
```

The method forName is as in the EncChecker.

newInstance() makes a new Object from testClass.

```
public class UnitTester{
    /** testClass is the class that contains the
        testcases */
    private Class testClass;
    /** testMethods is an list of the methods that
        contain the tests to be executed */
    private List testMethods;
    /** testObject is an instance of testClass. */
    private Object testObject;
    …
    public static String testTheClass(String name){
        try{
            UnitTester ut = new UnitTester();
            ut.setClass(name);
            ut.getMethods();
            return ut.performTests();
        }catch(Exception uups){
            return "Could not find class " + name;
        }
    }
}
```

17

The setClass method is a method in the UnitTest class.

It potentially throws a lot of different exceptions. ClassNotFound is thrown if the className is not found in the class path. The className must be a *fully qualified name*, that is, prefixed with the package name.

InstantiationException is thrown by the newInstance() method if testClass is abstract, is an interface, an Array, a primitive class (such as int), or has no argumentless constructor.

InstantiationException is thrown by the newInstance() method if testClass or its argumentless constructor is not accessible.

Note. In normal Java, new objects are created with an expression ”new ClassName(args)”. This syntax does not allow ClassName to be a String variable, nor a Class variable, it has to be a name of a class so that the compiler can check it when the code is compiled.

Finding the methods which start with 'test'

The method is very similar to the one we used in EncCheck to find all the fields.

getMethods returns an array of all methods in the class.

Each method can be asked about its name using getName.

All methods which starts with 'test' are stored in the List testMethods.

```
private List<Method> testMethods;

private void getMethods(){
    testMethods = new ArrayList<Method>();
    try{
        Method[] allMethods = testClass.getMethods();
        for (Method m : allMethods){
            if ( m.getName().startsWith("test") ){
                testMethods.add(m);
            }
        }
    }catch(SecurityException whatWasThat){
    }
}
```

18

This method here is in essence a filter which takes as input all methods in the class, and as output gives all methods which starts with "test".

Note how the variable testMethods is declared to be of the interface type list, but instantiated to be an instance of ArrayList. There is no deeper reason for this, but to give an example of how such interfaces are used.

Calling the test methods

The key method here is invoke. It takes two arguments:

- the object on which to call the method (the object that will be *this* in the method call)
- an array of Objects to be given as arguments to the method.

So a method normally called as:

myObj.m(args)

can be called using reflection as:

m.invoke(myObj, args)

instanceof is a Java operator, which returns true if the object on the left hand side is an instance of the class on the right hand side.

```
private String performTests(){
    String result = "Test of " + testClass.getName() + "\n";
    for (Method m: testMethods){
        try{
            m.invoke(testObject,new Object[0]);
            result += m.getName() + " OK\n";
        }catch(IllegalAccessException ignore){}
        catch(IllegalArgumentException ignore){}
        catch(InvocationTargetException uups){
            Throwable target = uups.getTargetException();
            if ( target instanceof TestClass.CheckFailure)
                result += m.getName() +" Check failed: "
          + ((TestClass.CheckFailure)target).msg + "\n";
    else
                result += m.getName() + " Exception: "+
                        target.toString() + "\n";
        }
    }
    return result;
}
```

19

The invoke method will throw an IllegalArgumentException if the elements in the array do not match in type and position with the parameters declared in the method.

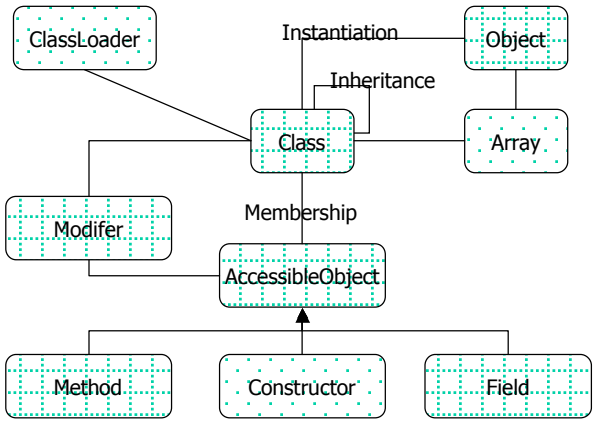
The execution of the method might throw all kinds of exceptions which we cannot know about. These are encapsulated into an InvocationTargetException, which basically means that the reflective call was all right, but the method failed because of another problem.

One source of failure is that the check method threw a CheckFailure exception. Those failures we want to deal with and report that the check failed. One can get to the original exception by the getTargetException method.

The if statement checks to see if the target (that is, the real problem) is the CheckFailure exception. In that case we report that the check failed. Otherwise the problem was something else, and we report that another exception caused the problem.

Please observe, in the code on the net, the error messages are more elaborated than shown above.

The Java meta model



Dotted classes have not been used in these slides, squared ones have.

20

The ClassLoader object we have not examined, but it was briefly mentioned in connection with the lecture on packages. It is the class loader which is responsible for getting classes from the file system into the virtual machine. The standard class loader is the one which uses the CLASSPATH variable to control this.

We will not look further on the class Array. It provides static methods to dynamically create and access Java arrays.

The class Class

The class Class represents four different types of java objects, classes, interfaces, primitive types, and array types.

One can ask a Class what kind of class it is by using the boolean methods: isArray(), isPrimitive(), isInterface() – if all return false, it is a class.

The class can tell what its super class is (getSuperClass) and which interfaces it implements (getInterfaces).

The class can tell which inner classes it has (getClasses), and an inner class can tell what class it is inner in (getDeclaringClass).

A class can tell which fields, methods, and constructors it contains.

The class Class is part of java.lang, and need not be imported to be used.

There are several ways of retrieving a Class object, i.e. the Class object for Person can be obtained using :

```
Class c1 = hans.getClass();
Class c2 = Class.forName("Person");
Class c3 = Person.class;
```

21

Note: When using reflection remember that not all operations make sense on all 4 types of Class objects. For instance it will not make much sense to call getConstructors() on an Interface or getFields() on an Array.

Field & AccessibleObject

The class Field represents field variables in a class, both static and non-static fields.

- Class getDeclaringClass()
- int getModifiers()
- String getName()
- Class getType()

- Object get(Object obj)
- boolean getBoolean(Object obj)
- X getX(Object obj)

- void set(Object obj, Object value)
- void setBoolean(Object obj, boolean z)
- void setX(Object obj, X d)

X can be any primitive type, but that would be too many methods to fit on a slide.

AccessibleObject is an abstract super class for Field, Method, & Constructor.

Its purpose is to allow us to mark a reflected entity to suppress access control.

- boolean isAccessible()
- void setAccessible(boolean flag)

There is a security control mechanism which can be set in the virtual machine, to prevent the manipulation of this flag.

22

The object passed as argument to get is necessary, because the Field object is a Field in a class, not a field in a specific object.

Thus, if the Field object f represents the field name of type String in a class Person, then to find the name of the object representing a specific person p, we find that through the call f.get(p). The more obvious call might have been p.getField(f), but that would mean that all the reflective method getField would have been placed in class Object.

With the f.get(p) design, we can separate the reflective methods from the plain methods.

The classes Method & Constructor

The class Method represents a method in a class.

- Class getDeclaringClass()
- Class[] getExceptionTypes()
- int getModifiers()
- String getName()
- Class[] getParameterTypes()
- Class getReturnType()
- Object invoke(Object obj, Object[] args)

The class Constructor represents a constructor in a class. It has nearly the same methods as Method:

- Class getDeclaringClass()
- Class[] getExceptionTypes()
- int getModifiers()
- String getName()
- Class[] getParameterTypes()
- Object newInstance(Object[] initargs)

The difference being that it has no return type, and that the invoke method is named newInstance.

23

The arguments to a method call is an Object array, but what to do if the method expects two int's, as in a method declared as

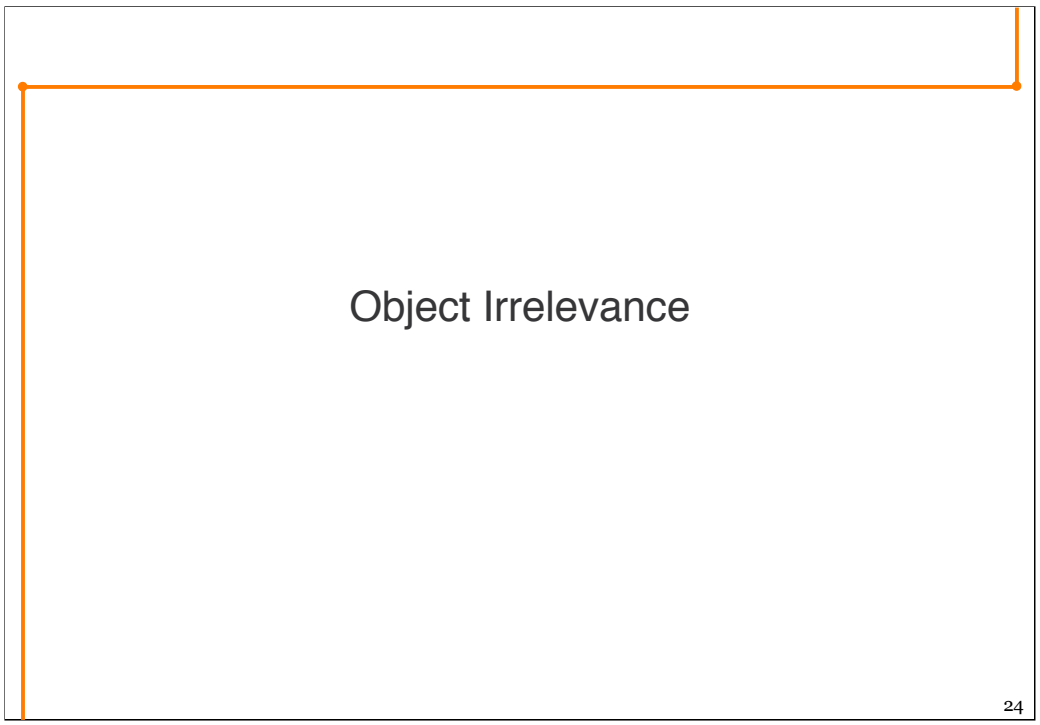
```
class InvokeTest{
    int c = 8;
    int sum(int a, int b){ return a+b+c;}
}
```

If we want to invoke the sum method on an object o of type InvokeTest with the arguments 2 and 5, it must be done as:

```
Method m = ... code that finds the sum method...;
Object[] arguments = new Object[]{new Integer(2), new Integer(5)};
Integer resultInteger = ( (Integer)m.invoke(o, arguments) );
int result = resultInteger.intValue();
```

Thus, primitive arguments must wrapped in their corresponding wrapper types. They are automatically unwrapped when calling sum. Similarly, the result of sum, which is an int, is automatically wrapped in an Integer.

Also note, it takes 20-100 times longer to call a method using reflection than calling it directly.


Object Irrelevance
24

Object Irrelevance

```
class Car {
  ...
}

class Color {
  ...
}

class People {
  ...
}
```

25

Object Irrelevance (cont'd)

- Objects model the real world
- Classes form sets of those models.
- Standard Object equality comes for free.
- Structural equality must be programmed!

Problem:

- Color red1 = new Color ()
- Color green = new Color ()
- Color red2 = new Color ()

red1 and red2 are not standard equal (red1 = red2 is not true)

- User responsible for maintaining the object references.

Object Irrelevance (cont'd)

- Alternative idea: Represent objects as classes.

Interface COLOR {...}

Class Red implements COLOR;
Class Green implements COLOR;
Glass Blue implements COLOR;

- Compiler prevents you from declaring the same color twice.
- All objects of the same class are equal (thus they are irrelevant).

```
Color red1 = new Red ();
Color red2 = new Red ();
red1.eq (red2)                    results in true.
```


Annotations
28

Introducing annotations

- Inspecting metadata can be done using *reflection*
- Adding custom metadata can be done using *annotations*

Annotations are used by three types of tools:

- General tools
  - I.e. compilers, JavaDoc
- Specific tools
  - I.e code generators
- Introspectors
  - That access annotations at runtime using reflection

29

Annotations are a new part of the Java language introduced with Java 5.0

Their purpose is to give a way of adding semantic metadata to programs. In earlier versions of Java there was no formal way of adding metadata to a program, so tools depending on metadata had to depend on solutions as naming conventions (as in testmethods in unit testing) and tags used in comments (as in JavaDoc)

Types of annotations

There are three types of annotations:

- Marker annotations
  - Have no parameters

```
@Test
public void myTestMethod() {...}
```

- Single member annotations
  - Have a single unnamed parameter

```
@Author("John Doe")
public class MyClass {...}
```

- Normal annotations
  - Have several members

```
@Author(company="NN Inc.",
  @Name(firstname="John",
                    lastname="Doe"))
interface MyInterface {...}
```

30

The parameters or members of annotations can be primitive types (int, boolean etc.), Strings, Class types, enumerations, other annotations or arrays of these.

All of the following Java elements can be annotated:

Packages, interfaces, classes, constructors, methods, parameters, local variables, enumerations and annotations.

In order to use annotations the java.lang.annotation package has to be imported.

Predefined annotations

@Retention has a RetentionPolicy defining how long to retain the annotations.

Possible values are:

SOURCE
discarded by compiler
CLASS (= default value)
retained by compiler, not loaded by VM
RUNTIME
retained by compiler and loaded runtime

If annotations are to be accessed using reflection the RetentionPolicy has to be set to RUNTIME.

@Target defines what types of Java elements can be annotated by this annotation.

Possible values are:

TYPE (class, interface or enum)
ANNOTATION_TYPE
CONSTRUCTOR
METHOD
PARAMETER
FIELD
LOCAL_VARIABLE
PACKAGE

31

There are several other predefined annotations that we won't cover here. If you want to read more about them look in the documentation for java.lang.annotation

The two mentioned annotations are both annotations on annotations. Others like i.e @Deprecated can be used on other Java elements than annotations.

When setting the retention policy and target of annotations the syntax is:

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
```

Declaring annotations

A new keyword @interface is used for declaring annotations. An annotation has a name and possibly some member methods.

```
public @interface Marker { }
```

Members can be given default values using the default keyword. If no default value is given the value must be specified when the annotation is applied.

```
public @interface SingleMember {
  String value();
}
```

So all of the following are correct:

```
@MyAnno(someValue=2, otherValues={}, yetAValue="Hi" )
@MyAnno(someValue=17, otherValues={"a"})
@MyAnno(otherValues={"a","b","c"})
```

```
@Retention(value=RUNTIME)
@Target(value=METHOD)
public @interface MyAnno {
  int someValue() default 0;
  String[] otherValues();
  String yetAValue() default "[blank]";
}
```

32

There are some restrictions on annotation declarations:

- No extends clause is permitted. (Annotation types automatically extend a new marker interface, java.lang.annotation.Annotation.)
- Methods must not have any parameters.
- Methods must not have any type parameters (in other words, generic methods are prohibited).
- Method return types are restricted to primitive types, String, Class, enum types, annotation types, and arrays of these types.
- No throws clause is permitted.

If an annotation with just one member called value() is declared it can be accessed using the shorthand notation:

@SingleMember("Hi")

instead of

@SingleMember(value ="Hi")

Applying annotations

Annotations can be applied at several levels, and several different annotations can be applied at each level.

Arrays of values are entered in brackets {} with commas between the values.

```
@ClassLevelAnnotation(arg1="val1",
  arg2={"arg2.val1","arg2.val2"})
public class AnnotationExample {

    @FieldLevelAnnotation()
    public String field;

    @ConstructorLevelAnnotation()
    public AnnotationExample() {
            // code
    }

   @MethodLevelAnnotationA("val")
   @MethodLevelAnnotationB(arg1="val1“,
            arg2="val2")
    public void someMethod(String string) {
            // code
    }
}
```

33

An annotation marker is placed immediately before the element it annotates, like a modifier.

An element can have several annotations listed in a row, but only one of each type (again like modifiers).

Accessing annotations

Annotations can be inspected at runtime using reflection.

The classes Class, Field, Method and Constructor all have the methods:

boolean **isAnnotationPresent**(MyAnno.class) returns true if an annotation of the given annotation type exists on this element

Annotation **getAnnotation**(MyAnno.class) returns the annotation of the given annotation type if any exists on this element

Annotation[] **getDeclaredAnnotations**() Returns all annotations that are directly present on this element

Annotation[] **getAnnotations**() Returns all annotations that are present on this element, including those inherited

34

HelloWorldAnnotationTest

```
// The Annotation Type
@Retention(RetentionPolicy.RUNTIME)
@interface SayHi {
    public String value();
}

// The Annotated Class
@ SayHi("Hello, class!")
class HelloWorld {

    @ SayHi("Hello, field!")
    public String greetingState;

    @ SayHi("Hello, constructor!")
    public HelloWorld() {
    }

    @ SayHi("Hello, method!")
    public void greetings() {
    }
}
```

```
//The Annotation consumer
public class HelloWorldAnnotationTest
{
  public static void main (String[] args) throws Exception {
    Class<HelloWorld> clazz = HelloWorld.class;
    System.out.println( clazz.getAnnotation(SayHi.class ) );

    Constructor<HelloWorld> constructor =
          clazz.getConstructor((Class[]) null);
    System.out.println(
          constructor.getAnnotation(SayHi.class));

    Method method = clazz.getMethod("greetings");
    System.out.println(method.getAnnotation(SayHi.class));

    Field field = clazz.getField("greetingState");
    System.out.println(field.getAnnotation(SayHi.class));
  }
}
```

35

This simple example of declaring, applying and accessing annotations shows how an Annotation can be retrieved from different elements using reflection.

The getConstructor(Class[] parameterTypes) method returns the constructor on the class that has the given parameters. In this case we want the constructor with no parameters so parameterTypes is null.

Other reflection capabilities not discussed

In Java

- The class loader
- Java debugging interface
- Call stack inspection
- References
- Proxy

In other languages than Java

- Changing how method calls work
- Redefining the notion of a subclass
- Redefining how object initialization is done
- Redefining how fields are accessed
- Redefining how objects are addressed

36

The Java debugging interface is an API which allows you to set breakpoints, inspect local variables and a lot of cool stuff, so you can write your own individual debugger.

We will take a look at call stack inspection when we examine exceptions in a later lecture. In essence it allows us to see which method called this one.

One cannot interact with the garbage collector, one can only ask it to collect some garbage. However, one can tell it to ignore some kinds of references when it figures out if an object is dead or alive.

The Proxy object is a way in which one can make an object pretend it implements an interface, but in reality it just forwards each call to some other objects. This is very practical, and necessary if one wants to do a distributed object system from scratch.

If one want to go beyond the capabilities in Java, the languages Smalltalk and especially CLOS are the obvious choices.

Summary

From a programming point of view, reflection does not deal with things in the application domain, but with the internals of the programs themselves.

The reflection library is for very general abstractions. Not abstractions from the application domain, but abstractions of objects.

Annotations can add custom abstractions for specific tools to use.

Use with care. It is a powerful tool. It is meant for framework development, not for application development.