Object-oriented programming

Second semester

Lecture №5

Reflection

What is Reflection

- ***Reflection:*** *the process by which a program can observe and modify its own structure and behavior at runtime*.
- Based on RTTI (Run-Time Type Identification):
  - RTTI: allows programs to discover at runtime and use at runtime types that were not known at their compile time
  - Non-RTTI / Traditional approaches:
    - assume all types are known at compile time
    - Polymorphism in OO languages: is a particular case of very limited RTTI

Kinds of tasks specific to Reflection

- ***Inspection (introspection)****: analyzing objects and types to gather information about their definition and behavior*.
  - Find the run-time type information of an object
  - Find information about a type (supertypes, interfaces, members)
    - Dynamic type discovery
- ***Manipulation****: uses the information gained through inspection to change the structure/behavior:*
  - create new instances of new types discovered at runtime
  - dynamically invoke discovered methods
    - Late binding: the types and methods used by a program are not known at compile-time
  - The most one could imagine to do in a reflective language: restructure types and objects on the fly !

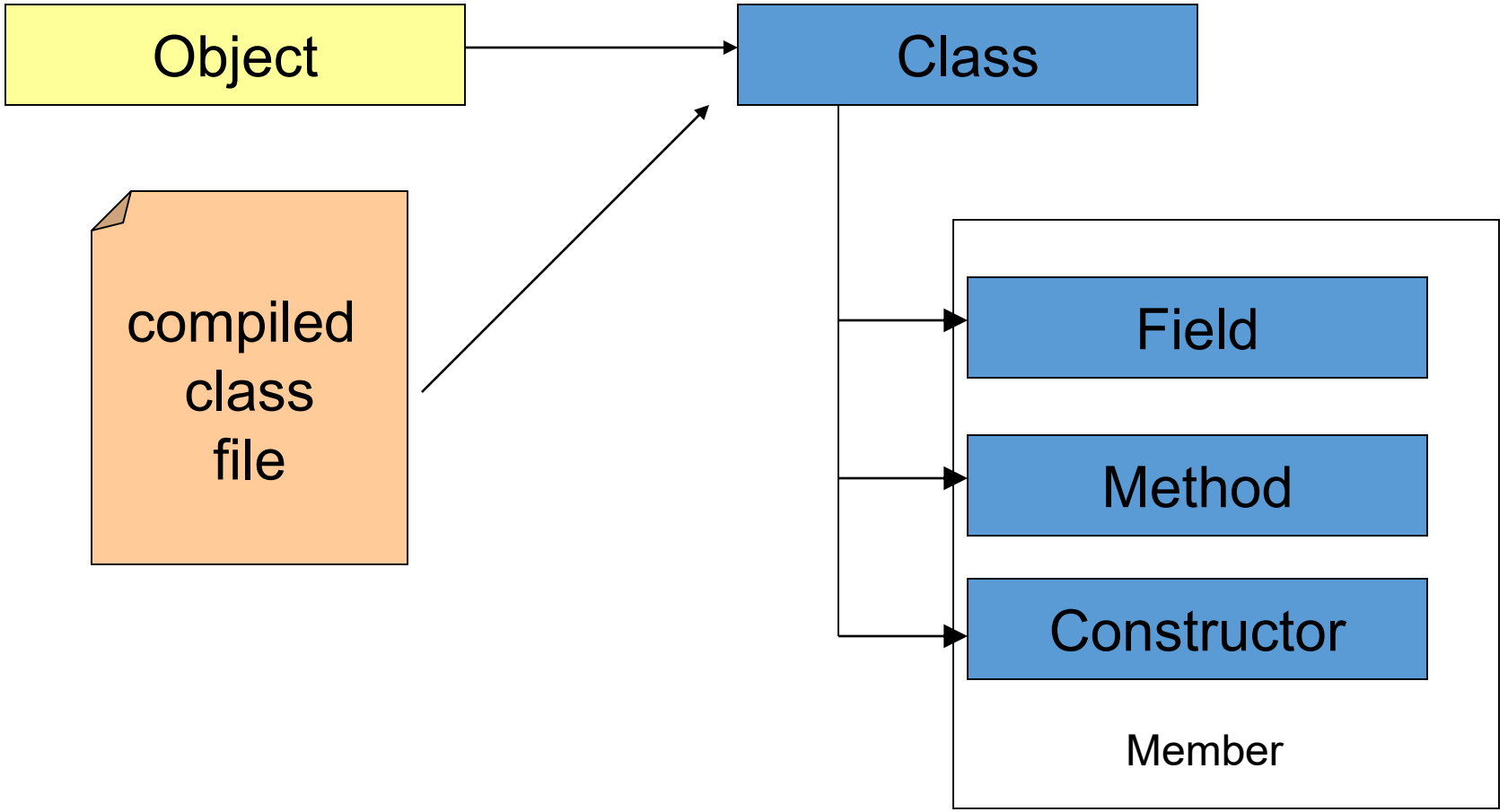
How is Reflection implemented

- Reflective capabilities need special support in language and compiler !
  - Java: java.lang.reflection
  - .NET: System.Reflection

Reflection case study: Reflection in Java

- `java.lang.reflect`
- Class `java.lang.Class<T>`
  - For every type of object, the JVM instantiates an immutable instance of java.lang.Class
  - Instances of the class Class represent classes and interfaces in a running Java application
  - It is the entry point for all of the Reflection API
  - Provides methods to examine the runtime properties of the object including its members and type information.
  - Provides the ability to create new objects of this type. It has no public constructor, but Class objects are constructed automatically by the JVM as classes are loaded
  - T is the type of the class modeled by this class object. Use Class<?> if the class is unknown.

The Reflection Logical Hierarchy in Java



Retrieving a Class object (1)

- **If there is an object (an instance) of this class available:**

- **Object.getClass():** If an instance of an object is available, then the simplest way to get its Class is to invoke Object.getClass().

```
Rectangle r;
…
Class<?> c = r.getClass();
…

Class<?> c= "foo".getClass();
```

Retrieving a Class object (2)

- **If the type is available but there is no instance:**

- **.class**: If the type is available but there is no instance then it is possible to obtain a Class by appending ".class" to the name of the type. This is also the easiest way to obtain the Class for a primitive type.

```
boolean b;
Class<?> c = b.getClass(); // compile-time error
Class<?> c = boolean.class; // correct
```

Retrieving a Class object (3)

- **If the class is available as compiled code in a file on the classpath:**

- **Class.forName():** If the fully-qualified name of a class is available, and *the file containing the compiled code is correctly put on the runtime classpath*, it is possible to get the corresponding Class using the static method Class.forName()

```
Class<?> cString = Class.forName("java.lang.String;");
```

Retrieving a Class object (4)

- **If the class is available as compiled code somewhere at a URL:**
- Each class object is constructed from bytecode by a java classloader
- The default class loader loads from the local file system only, directed by the classpath variable
- Other class loaders may load differently or from other locations. URLClassLoader loads from a URL, that may point to a directory or jar

```
URL[] urls = new URL[]{newURL("file:///home/pack.jar")};

URLClassLoader cl = new URLClassLoader(urls)

Class<?> c = cl.loadClass(“mypack.myclass”);
```

Inspecting a Class

- After we obtain a Class object `myClass`, we can:
- Get the class name

```
String s = myClass.getName() ;
```

- Get the class modifiers

```
int m = myClass.getModifiers() ;
bool isPublic = Modifier.isPublic(m) ;
bool isAbstract = Modifier.isAbstract(m) ;
bool isFinal = Modifier.isFinal(m) ;
```

- Test if it is an interface

```
bool isInterface = myClass.isInterface() ;
```

- Get the interfaces implemented by a class

```
Class [] itfs = myClass.getInterfaces() ;
```

- Get the superclass

```
Class<?> super = myClass.getSuperClass() ;
```

Discovering Class members

- fields, methods, and constructors
- java.lang.reflect.* :
  - Member interface
  - Field class
  - Method class
  - Constructor class

Class Methods for Locating Members

Member	Class API	List of members?	Inherited members ?	Private members ?
Field	getDeclaredField()	no	no	yes
	getField()	no	yes	no
	getDeclaredFields()	yes	no	yes
	getFields()	yes	yes	no
Method	getDeclaredMethod()	no	no	yes
	getMethod()	no	yes	no
	getDeclaredMethods()	yes	no	yes
	getMethods()	yes	yes	no
Constructor	getDeclaredConstructor()	no	N/A[1]	yes
	getConstructor()	no	N/A[1]	no
	getDeclaredConstructors()	yes	N/A[1]	yes
	getConstructors()	yes	N/A[1]	no

Working with Class members

- Members: fields, methods, and constructors
- For each member, the reflection API provides support to retrieve declaration and type information, and operations unique to the member (for example, setting the value of a field or invoking a method),
- java.lang.reflect.* :
  - “Member” interface
  - “Field” class: Fields have a **type** and a **value**. The java.lang.reflect.Field class provides methods for accessing type information and **setting and getting values** of a field on a given object.
  - “Method” class: Methods have **return values**, **parameters** and may throw **exceptions**. The java.lang.reflect.Method class provides methods for accessing type information for return type and parameters and **invoking** the method on a given object.
  - “Constructor” class: The Reflection APIs for constructors are defined in java.lang.reflect.Constructor and are similar to those for methods, with two major exceptions: first, constructors have no return values; second, the invocation of a constructor creates a new instance of an object for a given class.

Example: retrieving **public** fields

```
Class<?> c = Class.forName(“Dtest");

// get all public fields
Field[] publicFields = c.getFields();
for (int i = 0; i < publicFields.length; ++i) {
        String fieldName = publicFields[i].getName();
        Class <?> typeClass = publicFields[i].getType();
        System.out.println("Field: " + fieldName + " of type " +
                                                        typeClass.getName());
        }
```

Example

```
public class Btest
{
    public String aPublicString;
    private String aPrivateString;
    public Btest(String aString) {
            // …
    }
    public Btest(String s1, String s2) {
            // …
    }
    private void Op1(String s) {
            // …
      }
    protected String Op2(int x) {
            // …
    }
    public void Op3() {
            // …
    }
}
```

```
public class Dtest extends Btest
{
    public int aPublicInt;
    private int aPrivateInt;
    public Dtest(int x)
    {
            // …
    }

    private void OpD1(String s) {
            // …
    }

    public String OpD2(int x){
            // …
    }


}
```

Example: retrieving **public** fields

```
Class c = Class.forName(“Dtest");

// get all public fields
Field[] publicFields = c.getFields();
for (int i = 0; i < publicFields.length; ++i) {
        String fieldName = publicFields[i].getName();
        Class typeClass = publicFields[i].getType();
        System.out.println("Field: " + fieldName + " of type " +
                                                    typeClass.getName());
        }
```

```
Field: aPublicInt of type int
Field: aPublicString of type java.lang.String
```

Example: retrieving **declared** fields

```
Class c = Class.forName(“Dtest");

// get all declared fields
Field[] publicFields = c.getDeclaredFields();
for (int i = 0; i < publicFields.length; ++i) {
        String fieldName = publicFields[i].getName();
        Class typeClass = publicFields[i].getType();
        System.out.println("Field: " + fieldName + " of type " +
                                                typeClass.getName());
        }
```

```
Field: aPublicInt of type int
Field: aPrivateInt of type int
```

Example: retrieving **public** constructors

```
// get all public constructors

Constructor[] ctors = c.getConstructors();
for (int i = 0; i < ctors.length; ++i) {
    System.out.print("Constructor (");
    Class[] params = ctors[i].getParameterTypes();
    for (int k = 0; k < params.length; ++k)
            {
            String paramType = params[k].getName();
            System.out.print(paramType + " ");
            }
    System.out.println(")");
    }
```

```
Constructor (int )
```

Example: retrieving **public** methods

```
//get all public methods

Method[] ms = c.getMethods();
for (int i = 0; i < ms.length; ++i) {
    String mname = ms[i].getName();
    Class retType = ms[i].getReturnType();
    System.out.print("Method : " + mname + " returns " + retType.getName() + " parameters : (
    ");
    Class[] params = ms[i].getParameterTypes();
    for (int k = 0; k < params.length; ++k)
            {
            String paramType = params[k].getName();
            System.out.print(paramType + " ");
            }
    System.out.println(") ");
    }
```

```
Method : OpD2 returns java.lang.String parameters : ( int )
Method : Op3 returns void parameters : ( )
Method : wait returns void parameters : ( )
Method : wait returns void parameters : ( long int )
Method : wait returns void parameters : ( long )
Method : hashCode returns int parameters : ( )
Method : getClass returns java.lang.Class parameters : ( )
Method : equals returns boolean parameters : (
java.lang.Object )
Method : toString returns java.lang.String parameters : ( )
Method : notify returns void parameters : ( )
Method : notifyAll returns void parameters : ( )
```

Example: retrieving **declared** methods

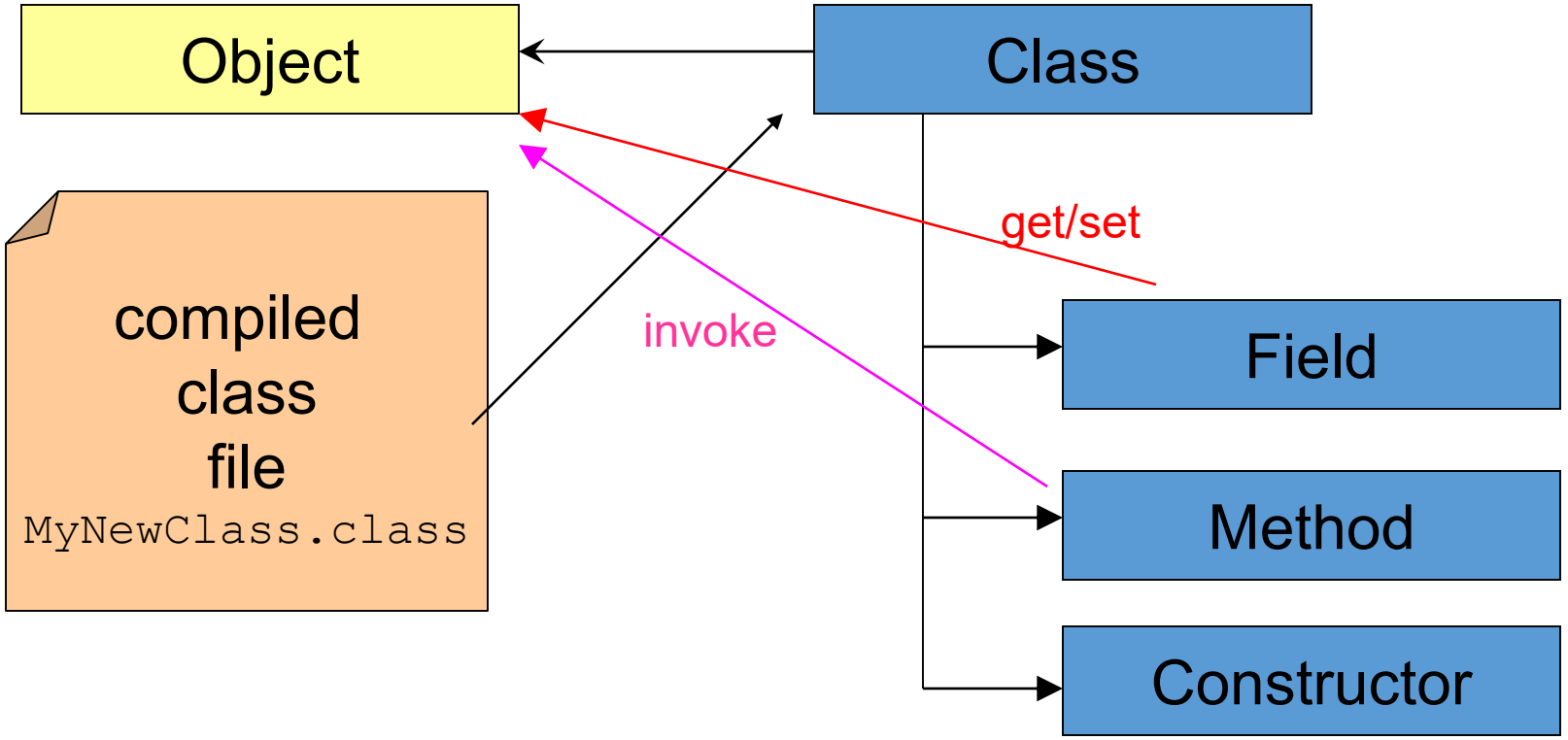
```
//get all declared methods

Method[] ms = c.getDeclaredMethods();
for (int i = 0; i < ms.length; ++i) {
    String mname = ms[i].getName();
    Class retType = ms[i].getReturnType();
    System.out.print("Method : " + mname + " returns " + retType.getName() + " parameters : (
    ");
    Class[] params = ms[i].getParameterTypes();
    for (int k = 0; k < params.length; ++k)
                {
                String paramType = params[k].getName();
                System.out.print(paramType + " ");
                }
    System.out.println(") ");
    }
```

```
Method : OpD1 returns void parameters : ( java.lang.String )
Method : OpD2 returns java.lang.String parameters : ( int )
```

Using Reflection for Program Manipulation

- Previous examples used Reflection for Introspection only
- Reflection is a powerful tool to:
  - Creating new objects of a type that was not known at compile time
  - Accessing members (accessing fields or invoking methods) that are not known at compile time

Using Reflection for Program Manipulation



Creating new objects

- Using Default Constructors
  - java.lang.reflect.Class.newInstance()

```
Class c = Class.forName("java.awt.Rectangle") ;
Rectangle r = (Rectangle) c.newInstance() ;
```

- Using Constructors with Arguments
  - java.lang.reflect.Constructor. newInstance(Object... initargs)

```
Class c = Class.forName("java.awt.Rectangle") ;
Class[] intArgsClass = new Class[]{ int.class, int.class } ;
Object[] intArgs = new Object[]{new Integer(12),new Integer(24)} ;
Constructor ctor = c.getConstructor(intArgsClass) ;
Rectangle r = (Rectangle) ctor.newInstance(intArgs) ;
```

Example

```
String className = "java.lang.String";

Class<?> c;

try {
 c = Class.forName(className);
 Class<?>[] stringArgsClass = new Class[] { String.class };
 Object[] stringArgs = new Object[] { new String("abc") };
 Constructor<?> ctor = c.getConstructor(stringArgsClass);

 Object something = ctor.newInstance(stringArgs);
 System.out.println(something.getClass().getName());
 System.out.println(something);
 } catch (Exception e) {
       e.printStackTrace();

}
```

Accessing fields

- Getting Field Values

```
Rectangle r = new Rectangle(12,24) ;
Class c = r.getClass() ;
Field f = c.getField("height") ;
Integer h = (Integer) f.get(r) ;
```

- Setting Field Values

```
Rectangle r = new Rectangle(12,24) ;
Class c = r.getClass() ;
Field f = c.getField("width") ;
f.set(r,new Integer(30)) ;
// equivalent with:  r.width=30
```

Invoking methods

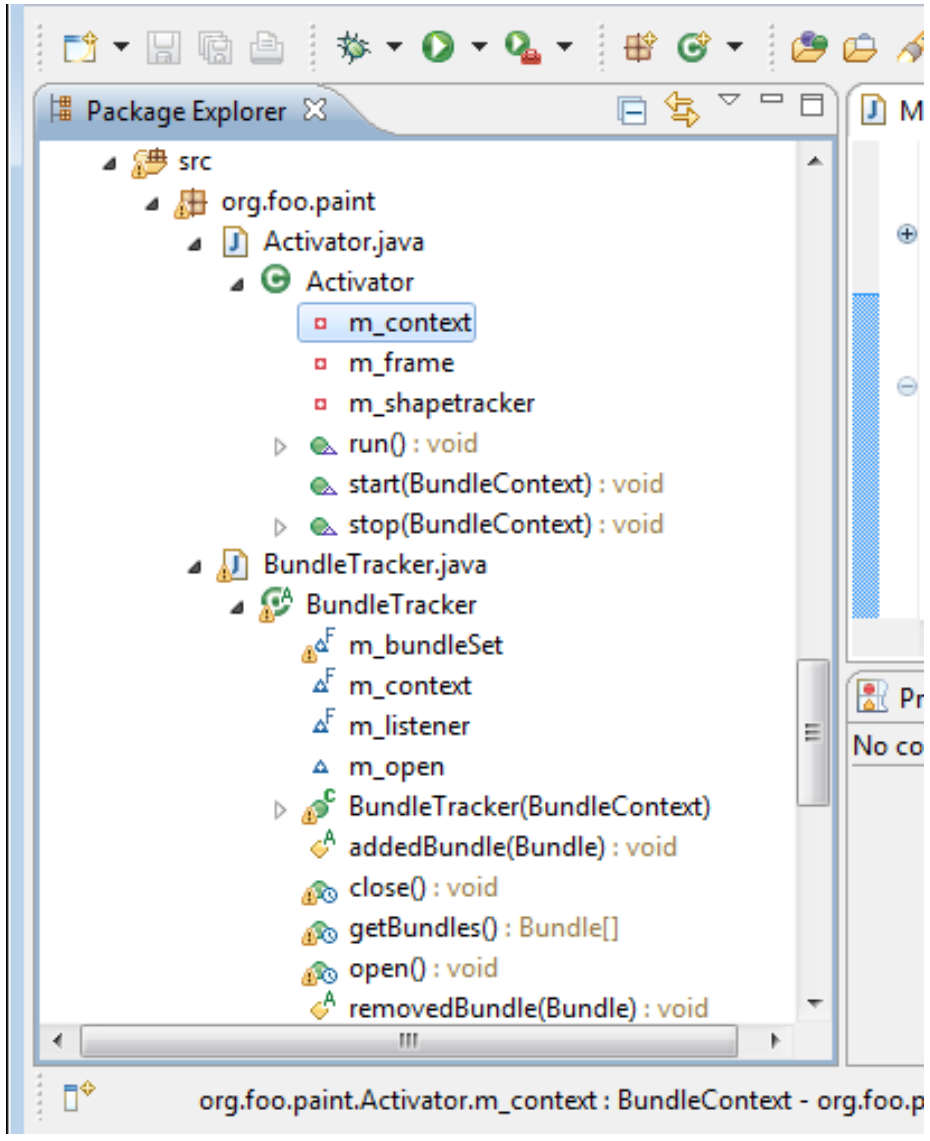
```
String s1 = “Hello ” ;
String s2 = “World” ;
Class c = String.class ;
Class[] paramtypes = new Class[] { String.class } ;
Object[] args = new Object[] { s2 } ;
Method concatMethod = c.getMethod(“concat”,paramtypes) ;
String result = (String) concatMethod.invoke(s1,args) ;
// equivalent with result=s1.concat(s2);
```

```
String className = "java.lang.String";
String methodName="length";

Class<?> c;

c = Class.forName(className);
Class<?>[] stringArgsClass = new Class[] { String.class };
Object[] stringArgs = new Object[] { new String("abc") };
Constructor<?> ctor = c.getConstructor(stringArgsClass);

Object something = ctor.newInstance(stringArgs);
System.out.println(something.getClass().getName());

Class<?>[] paramtypes = new Class[] {};
Object[] argsM = new Object[] {};
Method lengthMethod = c.getMethod(methodName, paramtypes);


System.out.println(lengthMethod.invoke(something, argsM));
```

Accessible Objects

- Can request that Field, Method, and Constructor objects be “accessible.”
  - Request granted if no security manager, or if the existing security manager allows it
- Can invoke method or access field, even if inaccessible via privacy rules !
- AccesibleObject Class: the Superclass of Field, Method, and Constructor
- boolean isAccessible( )
  - Gets the value of the accessible flag for this object
- static void setAccessible(AccessibleObject[] array, boolean flag)
  - Sets the accessible flag for an array of objects with a single security check
- void setAccessible(boolean flag)
  - Sets the accessible flag for this object to the indicated boolean value

Uses of Reflection - Examples



Example 1:
Class Browsers

Uses of Reflection - Examples

- **Serialization/deserialization in java:** the mechanisms of *writing the state of an object into a stream* and later retrieving the object from the stream
- Serialization could not be implemented without reflection

```
Student s1 =new Student(211,“John");

FileOutputStream fout=new FileOutputStream("f.txt");
ObjectOutputStream out=new ObjectOutputStream(fout);

out.writeObject(s1);
```

Uses of Reflection

- Extensibility Features :
  - An application may make use of external, user-defined classes by creating instances of extensibility objects using their fully-qualified names.
- Class libraries that need to understand a type’s definition
  - Typical example = Serialization
- Class Browsers and Visual Development Environments
  - A class browser needs to be able to enumerate the members of classes. Visual development environments can benefit from making use of type information available in reflection to aid the developer in writing correct code.
- Debuggers and Test Tools
  - Debuggers need to be able to examine private members on classes. Test harnesses can make use of reflection to systematically call a discoverable set APIs defined on a class, to insure a high level of code coverage in a test suite.

Drawbacks of Reflection

- If it is possible to perform an operation without using reflection, then it is preferable to avoid using it, because Reflection brings:
- **Performance Overhead**
  - Because reflection involves types that are dynamically resolved, certain Java virtual machine optimizations can not be performed. Consequently, reflective operations have slower performance than their non-reflective counterparts
- **Security Restrictions**
  - Reflection requires a runtime permission which may not be present when running under a security manager.
- **Exposure of Internals**
  - Since reflection allows code to perform operations that would be illegal in non-reflective code, such as accessing private fields and methods, the use of reflection can result in unexpected side-effects, which may render code dysfunctional and may destroy portability. Reflective code breaks abstractions and therefore may change behavior with upgrades of the platform.

Conclusions

- Reflective capabilities need special support at the levels of language (APIs) and compiler
- Language (API) level:
  - Java: java.lang.reflection
  - Very similar hierarchy of classes supporting reflection (Metaclasses)
- Compiler level:
  - Speciffic type informations are saved together with the generated code (needed for type discovery and introspection)
  - The generated code must contain also code for automatically creating instances of the Metaclasses every time a new type is defined in the application code