301AA - Advanced Programming

Lecturer: **Andrea Corradini**
andrea@di.unipi.it
http://pages.di.unipi.it/corradini/

AP-08: *Reflection in Java*

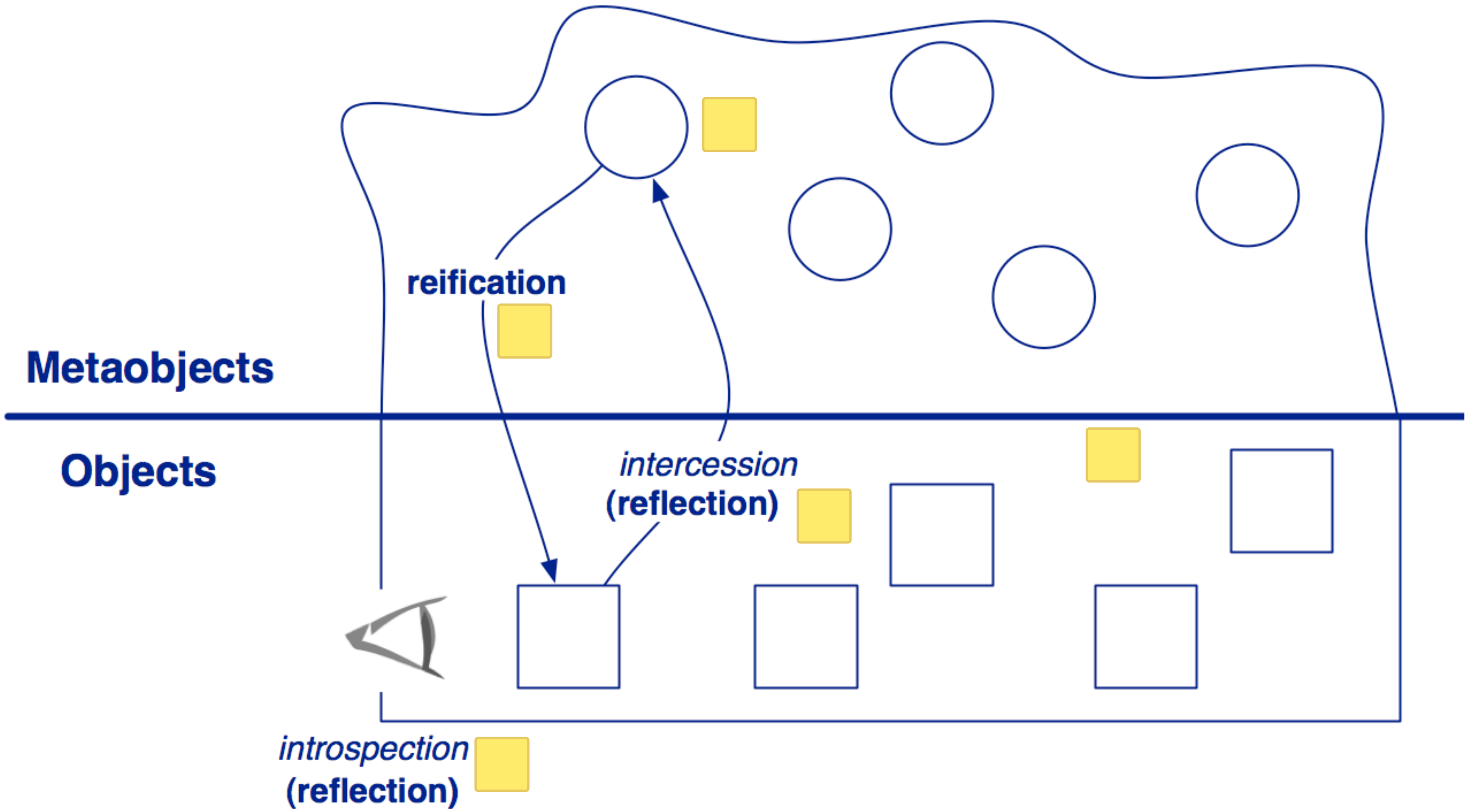
Overview

- Reflection in Programming Languages: pros & cons
- Reflection in Java
  - Class objects
  - Retrieving members of a class
  - Invoking methods and constuctors, accessing fields
  - Accessibility

➔ Tutorial: The Reflection API of Java
https://docs.oracle.com/javase/tutorial/reflect/index.html

Reflection

- **Reflection** is the ability of a program to manipulate as data something representing the state of the program during its own execution.
- A system may support reflection at different levels: from simple information on types to reflecting the entire structure of the program
- Another dimension of reflection is if a program is allowed to **read** only, or also to **change** itself
- **Introspection** is the ability of a program to *observe* and therefore *reason* about its own state
- **Intercession** is the ability for a program to *modify* its own execution state or *alter its own interpretation* or meaning
- Both aspects require a mechanism for encoding execution state as data: providing such an encoding is called **reification**.


reification
Metaobjects
Objects
intercession
(reflection)
introspection
(reflection)

Structural and behavioral reflection

- **Structural reflection** is concerned with the ability of the language to provide a complete *reification* of both
  - the *program* currently executed
  - as well as its *abstract data types*.
- **Behavioral reflection** is concerned with the ability of the language to provide a complete reification of
  - its own *semantics* and *implementation* (processor)
  - as well as the data and implementation of the *run-time system*.

Uses of Reflection

- **Class Browsers**
  need to be able to enumerate the members of classes
- **Visual Development Environments**
  can exploit type information available in reflection to aid the developer in writing correct code.
- **Debuggers**
  need to be able to examine private members on classes
- **Test Tools**
  can make use of reflection to ensure a high level of code coverage in a test suite
- **Extensibility Features**
  An application may make use of external, user-defined classes by creating instances of extensibility objects

Drawbacks of Reflection

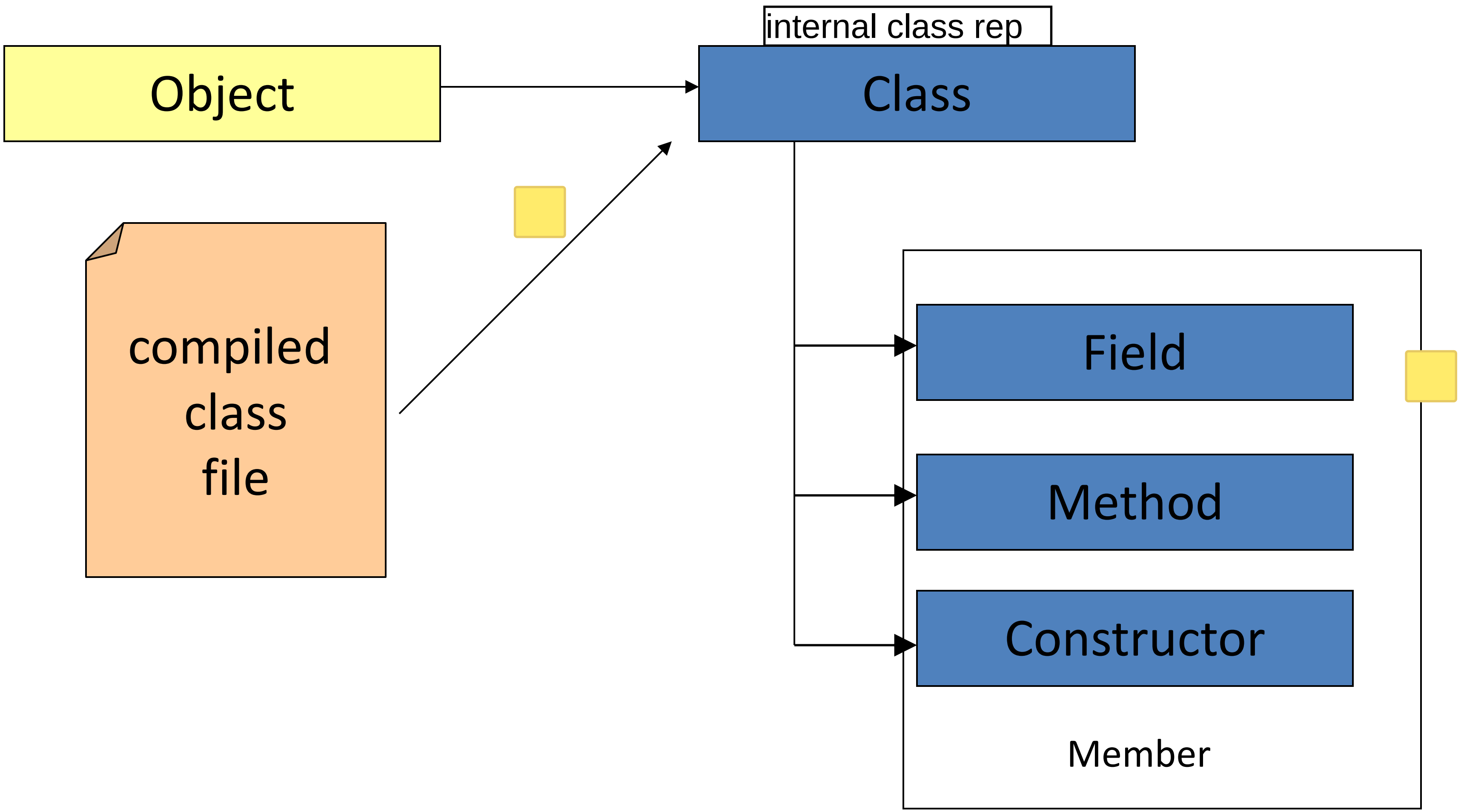
If it is possible to perform an operation without using reflection, then it is preferable to avoid using it, because Reflection brings:

- **Performance Overhead**
  Reflection involves types that are dynamically resolved, thus optimizations can not be performed, and 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. This affects code which has to run in a restricted security context, such as in an Applet.
- **Exposure of Internals**
  Reflective code may access internals (like private fields), thus it breaks abstractions and may change behavior with upgrades of the platform, destroying portability.

Reflection in Java

- Java supports **introspection** and **reflexive invocation**, but not code modification.
- For every type (*primitive*, *loaded* or *synthesized*), the JVM maintains an associated object of class **`java.lang.Class`**
- This object "reflects" the type it represents
- It is the "entry point" for reflection. All relevant information about the type can be obtained from it:
  - Class name & modifiers
  - Superclass & Interfaces implemented
  - Methods, fields, constructors, etc.
- API: **`java.lang.reflect`**

The Reflection Logical Hierarchy in Java



Retrieving Class Objects

- Use method `Object.getClass()`
- Examples:
  - `Class c = "foo".getClass(); // String`
  - ```
    byte[] bytes = new byte[1024];
    Class c = bytes.getClass(); //byte array
    ```
  - ```
    Set<String> s = new HashSet<String>();
    Class c = s.getClass(); // HashSet
    ```
- Use field **`.class`** of a type (also primitive)
  - `Class c = String.class;`
  - `Class c = boolean.class;`
  - `Class c = int[][][].class;`

Retrieving Class Objects (2)

- Use method `Class.`**`forName(String)`**
- Examples:
  - `Class c = Class.forName("java.util.List”);`
  - `Class c = Class.forName("[D"); // double[]`
  - `Class c = Class.forName("[[Ljava.lang.String;");`

Class object

```
class Class<T> ... {
  static Class<?>      forName( String name ) throws ...        {...}
  Method[ ]  getMethods( )                                       {...}
  Method[ ]  getDeclaredMethods( )                               {...}
  Method     getMethod( String name, Class<?>... parTypes )      {...}
  Class<? super T>  getSuperclass( )                             {...}
  boolean    isAssignableFrom( Class<?> cls )                    {...}
  T          newInstance( ) throws ...                           {...}
  ... }
```

Java

- Instances of the class Class represent classes and interfaces in a running Java application.
- Class objects are constructed automatically by the JVM as classes are loaded
- They provide access to the information read from the class file

Class file structure

ClassFile {

u4 magic;	**0xCAFEBABE**
u2 minor_version; u2 major_version;	**Java Language Version**
u2 constant_pool_count; cp_info contant_pool[constant_pool_count–1];	**Constant Pool**
u2 access_flags;	**access modifiers and other info**
u2 this_class; u2 super_class;	**References to Class and Superclass**
u2 interfaces_count; u2 interfaces[interfaces_count];	**References to Direct Interfaces**
u2 fields_count; field_info fields[fields_count];	**Static and Instance Variables**
u2 methods_count; method_info methods[methods_count];	**Methods**
u2 attributes_count; attribute_info attributes[attributes_count];	**Other Info on the Class**

}

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() ;
```

Printing Class Infos

```
public static void showType(String className)
                throws ClassNotFoundException {
  Class thisClass = Class.forName(className);
  String flavor = thisClass.isInterface() ?
          "interface" : "class";
  System.out.println(flavor + " " + className);
  Class parent = thisClass.getSuperclass();
  if (parent != null) {
    System.out.println("extends " + parent.getName());
  }
  Class[] interfaces = thisClass.getInterfaces();
  for (int i=0; i<interfaces.length; ++i) {
    System.out.println("implements "+
                          interfaces[i].getName());
  }
}
```

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(String)	no	no	yes
	getField(String)	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	yes
	getConstructor(…)	no	N/A	no
	getDeclaredConstructors()	yes	N/A	yes
	getConstructors()	yes	N/A	no

Class Methods for locating Fields

- **getDeclaredField(String name):** Returns **a Field object** representing the field called **name.** Must belong to the class **this** and can be private.
- **getField(String name):** Returns **a Field object** representing the field called **name.** Must be public and can belong to a **superinterface or superclass**.
- **getDeclaredFields():** Returns **an array of Field objects reflecting all the fields declared** by the class or interface represented by this Class object. This includes public, protected, default (package) access, and private fields, but excludes inherited fields.
- **getFields():** Returns **an array containing Field objects reflecting all the accessible public fields** of the class or interface represented by this Class object.

Class Methods for locating Methods

- **getDeclaredMethod(String name, Class<?>... parameterTypes):** Returns **a Method object** corresponding to the specified method, declared in this class
- **getMethod(String name, Class<?>... parameterTypes):** Returns **a Method object** corresponding to the public specified method
- **getDeclaredMethods():** Returns **an array of Method objects reflecting all (public and private) the methods declared** by the class or interface represented by this Class object.
- **getMethods():** Returns **an array containing Method objects reflecting all the accessible public methods** of the class or interface represented by this Class object.

Class Methods for locating Constructors

- **getDeclaredConstructor (Class<?>... parameterTypes):** Returns **a Constructor object** that reflects the specified constructor of the class or interface represented by this Class object. The parameterTypes parameter is an array of Class objects that identify the constructor's formal parameter types, in declared order.
- **getConstructor(Class<?>... parameterTypes):** Returns a Constructor object that reflects the specified **public** constructor of the class represented by this Class object. The parameterTypes parameter is an array of Class objects that identify the constructor's formal parameter types, in declared order.
- **getDeclaredConstructors():** Returns **an array of Constructor objects** reflecting all the constructors declared by the class represented by this Class object. These are public, protected, default (package) access, and private constructors. The elements in the array returned are not sorted and are not in any particular order.
- **getConstructors():** Returns **an array containing Constructor objects reflecting all the accessible public constructors**

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, invoking a method, creating an object)
- **`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.

Working with Class members

– **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 differences:
  - constructors have no return values
  - the invocation of a constructor **creates a new instance** of an object for a given class

Example

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

```java
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

```
// get all public fields
   try{
        Class c = Class.forName("Dtest");
        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());
        }
   } catch (ClassNotFoundException e){
        System.out.println("Class not found...");
   }
```

```
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)

Generic methods: effects of erasure

- **getMethod(String name, Class<?>... parameterTypes): **Returns **a Method object** corresponding to the public specified method

```java
try {
  LinkedList<String> list = new LinkedList<String>( );
  Class c = list.getClass( );
  Method add = c.getMethod( "add", String.class );
} catch( Exception e ) {
  System.out.println( "Method not found" );
}
```

Java

```
Method not found
```

- Due to Java's **erasure semantics**, generic type information is not represented at run time

Generic methods: effects of erasure (2)

```java
try {
  LinkedList<String> list = new LinkedList<String>( );
  Class c = list.getClass( );
  Method add = c.getMethod( "add", Object.class );
} catch( Exception e ) {
  System.out.println( "Method not found" );
}
```

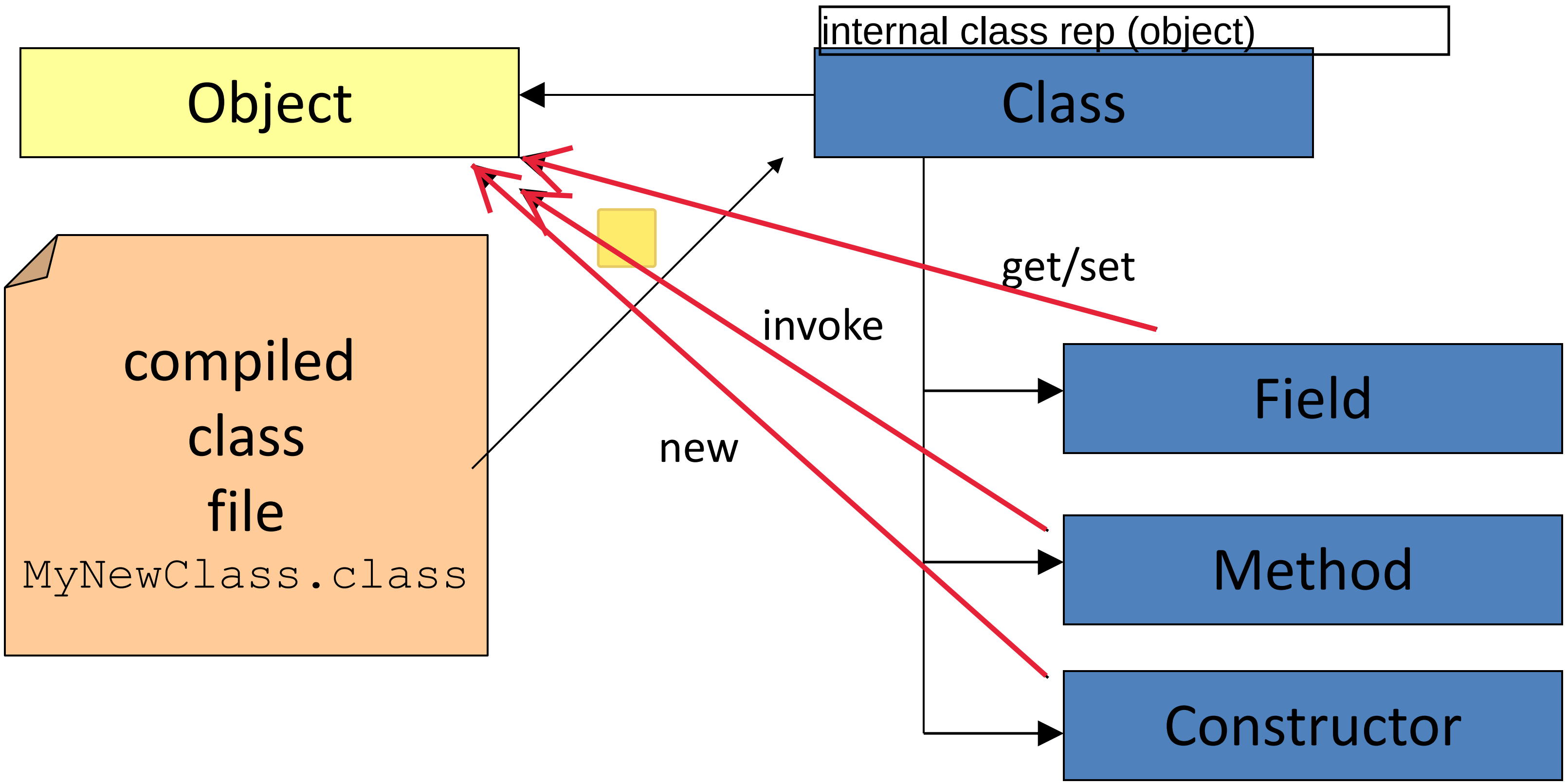
Java

```
// no exception
```

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.Class.newInstance()**

```
Rectangle r = new Rectangle();
```

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

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

```
Rectangle r = new Rectangle(12,24);
```

```
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);
```

Accessing fields

- Getting Field Values

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

- Setting Field Values

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

Invoking methods

```
String s1 = "Hello" ;
String s2 = "World" ;
// result = s1.concat(s2);
```

```
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) ;
```

Accessible Objects

- Certain operations are forbidden by privacy rules:
  - Changing a final field
  - Reading or writing a private field
  - Invoking a private method…
- Such operations fail also if invoked through reflection
- The programmer can request that Field, Method, and Constructor objects be "accessible."
  - Request granted if no security manager, or if the existing security manager allows it
- In this case you can invoke method or access field, even if inaccessible via privacy rules!
- `AccessibleObject` Class: the superclass of `Field`, `Method`, and `Constructor`

Accessible Objects (cont.)

- `AccessibleObject` provides the methods:
  - `boolean isAccessible( )`
    - Gets the value of the accessible flag for this object
  - `void setAccessible(boolean flag)`
    - Sets the accessible flag for this object to the indicated boolean value
  - `static void setAccessible( AccessibleObject[] array, boolean flag)`
    - Sets the accessible flag for an array of objects with a single security check

Accessing private fields

```
public static String getString( Object o ) {
  if ( o == null ) return "null";
  Class toExamine = o.getClass( );
  String state = "[";
  Field[ ] fields = toExamine.getDeclaredFields( );
  for ( int fi = 0; fi < fields.length; fi++ )
   try {
    Field f = fields[ fi ];
    if ( !Modifier.isStatic( f.getModifiers( ) ) )
     state += f.getName() + "=" + f.get( o ) + ", ";
   } catch ( Exception e ) { return "Exception"; }
  return state + "]";
}
```

Java

```
class Cell {
  private int value = 5;
  ...
}
```

```
Cell c = new Cell( );
String s = getString( c );
System.out.println( s );
```

Exception

java.lang.reflect Class Field

```
public Object get(Object obj)
     throws IllegalArgumentException, IllegalAccessException
```

Returns the value of the field represented by this Field, on the specified object. The value is automatically wrapped in an object if it has a primitive type.

The underlying field's value is obtained as follows:

- <omissis>

- If this Field object is enforcing Java language **access control**, and the underlying field is **inaccessible**, the method throws an **IllegalAccessException**. If the underlying field is static, the class that declared the field is initialized if it has not already been initialized.

Accessing private fields (2)

```java
public static String getString( Object o ) {
  if ( o == null ) return "null";
  Class toExamine = o.getClass( );
  String state = "[";
  Field[ ] fields = toExamine.getDeclaredFields( );
  for ( int fi = 0; fi < fields.length; fi++ )
    try {
      Field f = fields[ fi ];
      f.setAccessible( true );
      if ( !Modifier.isStatic( f.getModifiers( ) ) )
        state += f.getName() + "=" + f.get( o ) + ", ";
    } catch ( Exception e ) { return "Exception"; }
  return state + "]";
}
```

Suppress Java's access checking

Java

```java
class Cell {
  private int value = 5;
  …
}
```

```java
Cell c = new Cell( );
String s = getString( c );
System.out.println( s );
```

```
[value=5, ]
```

Exploiting Reflection: Unit Testing

```
class Cell {
  int value;
  Cell( int v ) { value = v; }
  int get( ) { return value; }
  void set( int v )
       { value = v; }
  void swap( Cell c ) {
    int tmp = value;
    value = c.value;
    c.value = tmp;
    }
}
```

```
class TestCell {
   void testSet( ) { ... }
   void testSwap( ) {
      Cell c1 = new Cell( 5 );
      Cell c2 = new Cell( 7 );
      c1.swap( c2 );
      assert c1.get( ) == 7;
      assert c2.get( ) == 5;
   }
}
```

Exploiting Reflection: Unit Testing (cont.)

```
public static void testDriver( String testClass ) {
   Class c = Class.forName( testClass );
   Object tc = c.newInstance( );
   Method[ ] methods = c.getDeclaredMethods( );

  for( int i = 0; i < methods.length; i++ ) {
   if( methods[ i ].getName( ).startsWith( "test" ) &&
       methods[ i ].getParameterTypes( ).length == 0 )
      methods[ i ].invoke( tc );
   }
}
```

A generic driver; the basic mechanism behind **JUnit**

From Modifiers to **Annotations**

- Modifiers in Java (*static, final, public,* ...) are *meta-data* describing properties of program elements
- Modifiers are reserved keywords, thus wired-in in the language
- Need for additional mechanisms for providing meta-data, without changing the language
- Annotations can be understood as (user-) definable modifiers

Structure of Annotations

- Annotations are made of
  - Annotation name
  - A finite number of *attributes*, i.e. "*name = value*" pairs, possibly none
- Syntax:
  - *@annName*   eg: **@Override**
  - *@annName{constExp}*
    shorthand for   *@annName{value=constExp}*
  - *@annName{name_1 = constExp_1, ..., name_k = constExp_k}*
- *constExp's* are expressions that can be evaluated at compile time
- Attributes have a type, thus the supplied values have to convertible to that type

Which elements can be annotated?

- Annotations can be applied to almost any syntactic element:
  - package declarations
  - classes (including enumeration types)
  - interfaces (including annotations)
  - fields and local variables
  - methods and constructors
  - parameters
  - (recently) any type use
- They can occur, in any number, together with other modifiers
- An annotation associates the name and set of indicated attributes to the modified

Some predefined annotations

- The Java compiler defines and recognizes a small set of *predefined annotations*. User defined annotations are ignored on compilation, but can be used by other tools.
- **@Override**. Makes explicit the intention of the programmer that the declared method overrides a method defined in a superclass. The compiler can issue a warning if no method is overridden.
- **@Deprecated**. Declares that the annotated element is not necessarily included in future releases of the Java API. Typically applied to methods, but also to classes and interfaces
- **@SuppressWarnings**. Instruct the compiler to avoid issuing warnings for the specified situations (e.g. ***all, cast, deprecation, divzero, overrides, unchecked, empty***,…). Example:

```
@SuppressWarnings({"deprecation","empty"})
void antiqueMethod () {
   OldClass.deprecatedMethod();
   ; // why not?
}
```

- ~~**@FunctionalInterface**. Declares an interface to be *functional*.~~

Define and use your own annotations

- Programmers can define new annotations, to be used
  - for documentation purposes of the source
  - to implement tools that process the content of the **.class** files generated by the compiler
  - to inspect the annotations placed on a class at runtime.
- The annotations have a declaration syntax similar to interfaces (but starting with **@interface**).
- Typically, an annotation type is an interface defining fields corresponding to the attributes.

Example: Annotation @InfoCode

```
@interface InfoCode {
        String author ();
        String date ();
        int ver () default 1;
        int rev () default 0;
        String [] changes () default {};
}
```

- Each method determines the name of an attribute and its type (the return type).
- A default value can be specified for each attribute (as for **ver**, **rev** and **changes**).
- Attribute types can only be **primitive**, **String**, **Class,** an **Enum**, an **Annotation**, or an array of those types.
- Additionally (like any interface) an @interface can contain constant declarations (with explicit initialization), internal classes and interfaces, enumerations, but rarely used.

Example: Annotation @InfoCode (2)

```
@interface InfoCode {
        String author ();
        String date ();
        int ver () default 1;
        int rev () default 0;
        String [] changes () default {};
}
```

- The annotation could then be applied to various program elements, as in this case:

```
@InfoCode(author="Beppe", date="10/12/07")
public class C {
   public static void m1() { /* ... */ }
   @InfoCode(author="Gianni",
          date="4/8/08", ver=1, rev=2)
   public static void m2() { /* ... */ }
}
```

Annotating annotations

Annotation definitions can be annotated in turn, to describe their meta-data. Some predefined meta-annotations:

- **@Target**. Constrains the program elements to which the annotation can be applied. The value type is **annotation.ElementType []**, an enum including **ANNOTATION_TYPE, CONSTRUCTOR, FIELD, LOCAL_VARIABLE, METHOD, PACKAGE, PARAMETER, TYPE_PARAMETER, TYPE_USE.**
- **@Retention**. Till when should the annotation be present? Three options (values of enum **RetentionPolicy**): **SOURCE, CLASS** (default), **RUNTIME**
- **@Inherited.** Marker annotation. The annotation is inherited by subclasses.

Recovering annotations through the Reflection API

- Annotations in class files can be exploited by appropriate tools for program analysis. Package **javax.annotation.processing** provides a Java API for writing such tools.
- Retrieval of annotations at runtime occurs through the Reflection API.
- Relevant classes in **java.lang.reflect** (and **java.lang.Class**) provide suitable methods for retrieving annotations.
- For example
  - **Annotation[] getAnnotations()**
    in class **Class**: returns an array of Annotation instances
  - **<T extends Annotation> T getAnnotation(Class<T> annotationClass)**
    in class **Method**: returns this element's annotation for the specified type if such an annotation is present, else null

A comprehensive example

```java
import java.lang.annotation.*;

@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE,ElementType.PACKAGE})
@interface InfoCode {
    String author ();
    String date ();
    int ver() default 1;
    int rev() default 0;
    String[] changes() default {};
}
@InfoCode(author="Gigi", date="8/12/2008")
public class TestAnno {
    @SuppressWarnings("unchecked")
    public static void main(String[] args) {
       Class c = TestAnno.class;
       System.out.print("I am: " + c.toString());
       InfoCode ic = (InfoCode)c.getAnnotation(InfoCode.class);
       if (ic != null)
           System.out.print(" v" + ic.ver() + "." + ic.rev()
                + " by " + ic.author());
       System.out.println();
    }
}      // prints:      I am: class TestAnno v1.0 by Gigi
```

Conclusions

- Reflective capabilities need special support at the levels of language (APIs) and compiler
- Language (API) level:
  - Java: java.lang.reflection
  - .NET: System.Reflection
  - Very similar hierarchy of classes supporting reflection (Metaclasses)
- Compiler level:
  - Specific type information is 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