Universitatea Tehnica din Cluj-Napoca
Departament Calculatoare

# Programming Techniques in Java

## Compositional Techniques
## &
## Reflection

T. Cioara, V. Chifu, C. Pop
2025

# Inheritance Technique

- **Stack inherits from ArrayList**

```
public class  ArrayList<T> {
    …
    // see if collection is empty
    public boolean isEmpty() { … }

    // return size of collection
    public int size() { …}

   // add element to the end
   public void add(T value) { … }

   // remove element at given index
   public T remove(int index) { … }

   // get element from index
   T get(int index) { … }
   … other class resources
}
```

```
public class Stack1<T> extends ArrayList<T> {
   public T push(T elem){
      T retObject;
      if(isFull()) retObject = null;
      else { add(elem); retObject = elem; }
      return retObject;
   }
   public T pop() {
      T retObject;
      if(isEmpty()) retObject = null;
      else {
         retObject = get(size()-1);
         remove(size()-1);
      }
      return retObject;
   }
   public T top() {
      T retObject;
      if(isEmpty()) retObject = null;
      else retObject = get(size()-1);
      return retObject;
   }
   public boolean isFull() { return false };

}
```

# Inheritance Technique

- **Stack inherits from ArrayList**
  - Stack structural component is inherited from ArrayList
  - Adding class specific methods push, pop, top
  - No data elements defined by the class Stack
  - All data elements are inherited from ArrayList
- Uses the inherited methods in the implementation of the Stack specific methods
  - But inherits all the other methods of ArrayList
  - Problem because now they can be used by Stack objects

```
stack.add(0, string);  // Add to bottom of stack
stack.remove(0);       // Remove from bottom of stack
stack.indexOf(Object); // Returns the index of the first occurrence
```

# Inheritance Technique

| Advantages | Disadvantages |
|---|---|
| • New implementation is easy, since most of it is inherited<br>• Less code<br>• Easy to modify or extend the implementation being reused<br>• Less overhead in execution, than the composition<br>• Allows using the new abstraction as an argument in an existing polymorphic method<br>• Better execution time | • A user (programmer) should study and understand the methods of the superclass<br>• Operations are more difficult to understand<br>• Breaks encapsulation, since it exposes a subclass to implementation details of its superclass<br>• "White-box" reuse, since internal details of superclasses are often visible to subclasses<br>• Subclasses may have to be changed if the implementation of the superclass changes<br>• Implementations inherited from superclasses can not be changed at runtime as composition blocks |

# Aggregation Technique

- **Stack uses ArrayList**

The new functionality is obtained by delegating functionality to one of the objects being composed

```
public class Stack2<T> {

   private ArrayList<T> stk;

   public Stack2() { stk = new ArrayList<>();}

   // behavior
   public T push(T o) {
     T retObject;
     if(isFull()) retObject = null;
       else {
         stk.add(o);
         retObject = o;
     }
     return retObject;
   }
   public T pop() {
     T retObject;
     if(stk.isEmpty()) retObject = null;
     else { retObject=stk.remove(stk.size()-1);}
     return retObject;
    }
}
```

# Aggregation Technique

- Stack uses ArrayList
  - Stack class - defines  private instance variable (stk) of type ArrayList
  - Strong composition when allocating the ArrayList object
  - Code reuse
  - Difficult work is delegated to ArrayList methods
  - Composition makes no explicit or implicit claims for substitutability.
  - Stack and ArrayList - entirely distinct entities
- Problems solved

```
    stk.add(0, string);  // not accessible
    stk.remove(0);       // not accessible
```

# Aggregation Technique

## Advantages

- Clearly shows all available operations for the abstraction that aggregates
- "Black-box" reuse, => good encapsulation
- Compositions are simple to be changed
- Better separates the two abstractions
- Contained objects are accessed by the containing class solely through their interfaces
- Fewer implementation dependencies
- The composition can be defined dynamically at run-time using polymorphism

## Disadvantages

- Longer code
- Resulting systems tend to have more objects
- Interfaces must be carefully defined in order to use many different objects as composition blocks

# Composition versus Inheritance

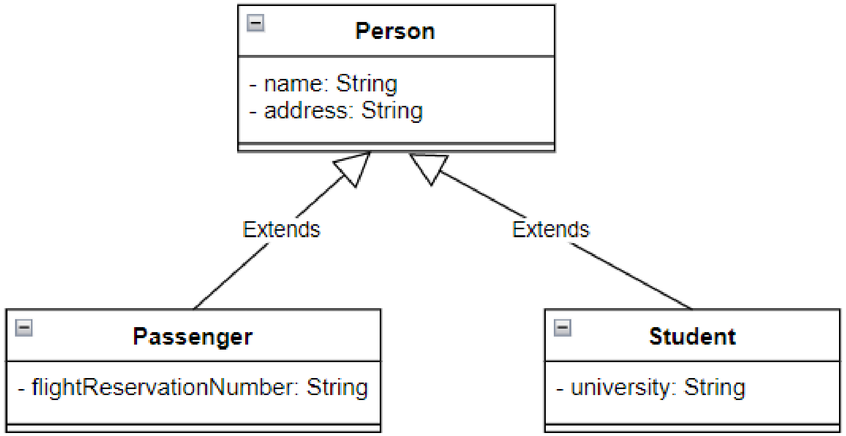
**Always Favor composition over inheritance**

- **Coad rules for identifying when inheritance should be used**

All should be true

| | |
|---|---|
| R1 | A subclass expresses "is a special kind of" and not "is a role played by a“ |
| R2 | An instance of a subclass never needs to become an object of another class |
| R3 | A subclass extends, rather than overrides the responsibilities of its superclass |
| R4 | A subclass does not extend the capabilities of what is merely a utility class |
| R5 | For a class in the actual Problem Domain, the subclass specialize a role, transaction or device |

# Coad rules usage example




**Answer to Coad questions:**

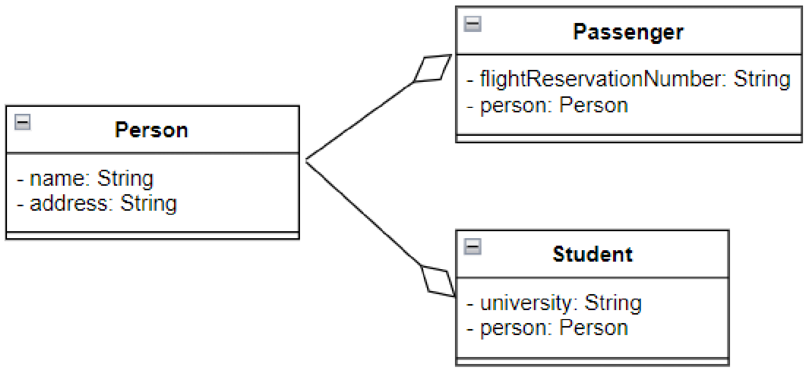
- **Q1:** “is a special kind of" and not "is a role played by a“  => **False**
- **Q2:** An instance of a subclass never needs to become an object of another class
  – An instance of a Person may change (in time) from Passenger to Student => **Fail**
- **Q3:** Extends, rather than overrides, the responsibilities of its superclass => **True**
- **Q4:** The subclasses are not extending a utility class => **True**
- **Q5:** For a class in the actual Problem Domain, the subclass specialize a role, transaction or device

  => **False** because a person is not a role, transaction or device

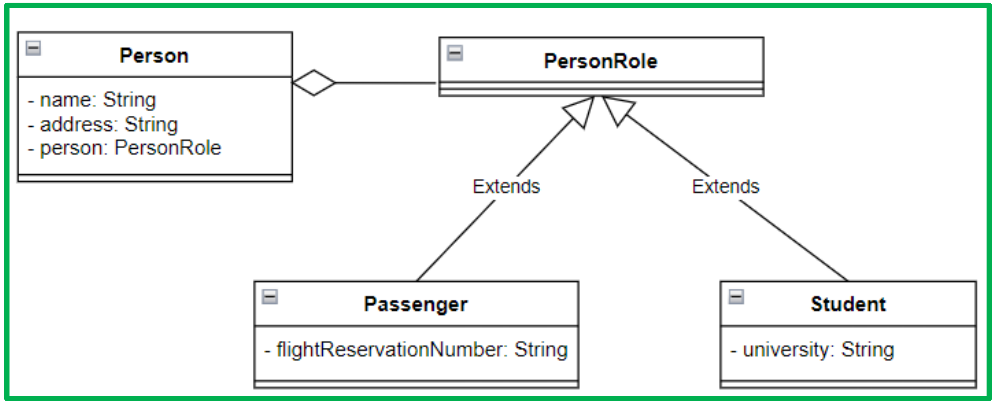
**Inheritance is not appropriate here**

# Coad rules usage example

**Incomplete solution**




**Good solution**

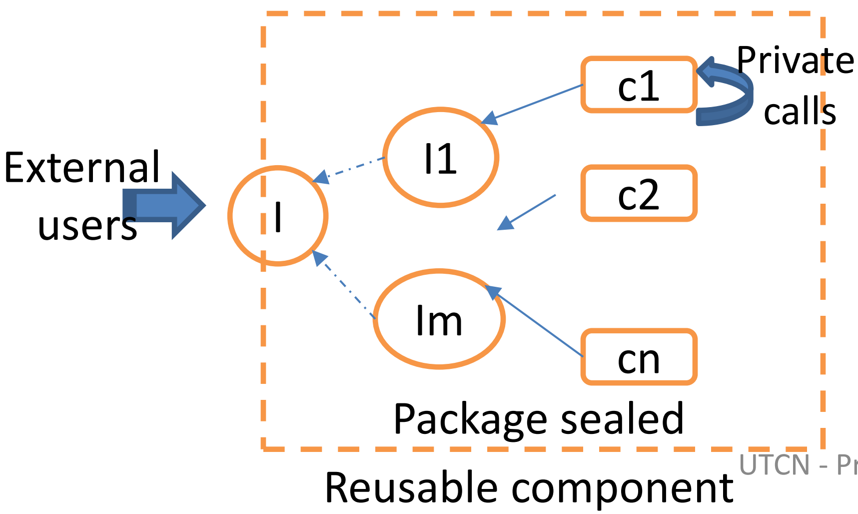



- **Q1:** Passenger and Student are special kinds of Person roles **=> True**
- **Q2:** A Passenger object stays a Passenger object; the same is true for a Student object **=> True**
- **Q3:** Extends, rather than overrides or nullifies, the responsibilities of its superclass => **True**
- **Q4:** The subclasses are not extending a utility class => **True**
- **Q5:** For a class in the actual Problem Domain, the subclass specialize a role, transaction or device => **True** because PersonRole is a role

**Inheritance is appropriate here**

# Sealed Classes and Interfaces

- Goal: restricting the set of subclasses
  – Limited solutions before java 15
  – Final class or package-private constructors
- Extended or implemented only by those permitted to do so
  – Should be as close as possible
  – Permitted subclass may be declared: final, sealed, non sealed
  – instanceof expression tests and open extensibility notion in java




```
public abstract sealed class Shape
  permits Circle, Rectangle, Square { ... }
```

```
interface I {}
sealed class C permits D, E {}
non-sealed class D extends C {}
final class E extends C {}

void test (C c) {
    if (c instanceof I) …}
```

# Reflection

- Overview
  - Ability of a running program to examine itself and change its actions depending on what it finds
  - Dynamic access/inspection to internal information for classes

**Advantages**

- Building flexible code that can be assembled at run time
- No required source code links between components

**Disadvantages**

- Security
- Code Mintenance
- Performance
- Issues when debugging reflection code for classes which are not accessible at compile time

# Reflection

- Motivation
  - Implementation of a graphical user interface which integrates
    - Java visual components
    - Visual components developed in house
    - Open-source visual components
    - Licensed 3rd party visual components
  - Each component provides a *setColor* method that takes a *java.awt.Color* parameter
  - The only common base class for them is *java.lang.Object*
  - The components cannot be referenced using a common type that supports the *setColor* method



Source: I. Forman, N. Forman, Java Reflection in Action, Manning Publications, 2005 – Chapter 1

# Reflection

- Scenario - What if we want to call ***setColor*** regardless of a component's concrete type?
  - **Option 1 - Refactor the components to implement a common interface declaring *setColor***
    - PROBLEM: the standard Java or 3rd party components can not be controlled => the option is not feasible!
  - **Option 2 – Implement an adapter for each component**
    - PROBLEM: Explosion in the number of classes to maintain, large number of objects in the system at runtime
  - **Option 3 – using *instanceof* and casting to discover concrete types at runtime**
    - PROBLEM: code with many conditionals and casts; code coupled with each concrete type -> difficult to add, remove or change components



Source: I. Forman, N. Forman, Java Reflection in Action, Manning Publications, 2005 – Chapter 1

# Reflection

- Scenario - What if we want to call ***setColor*** regardless of a component's concrete type?
  - **Option 4 – Use <u>reflection</u>**

Step 2: query class object for setColor method

```
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);
   }
}
```

Step 1: query object for its class

Step 3: call resulting method on target obj

Source: I. Forman, N. Forman, Java Reflection in Action, Manning Publications, 2005 – Chapter 1

**Notes**:
- Step 1 and Step 2 perform **introspection**
- Step 3 performs **dynamic invocation**

# Reflection

## Metadata is data about data

- Needs to have a representation of itself
- Metadata is organized into objects called meta-objects
- **Introspection** - runtime examination of meta-objects

## Metadata for a class is stored in **java.lang.Class**

- This is the entry point into reflection operations

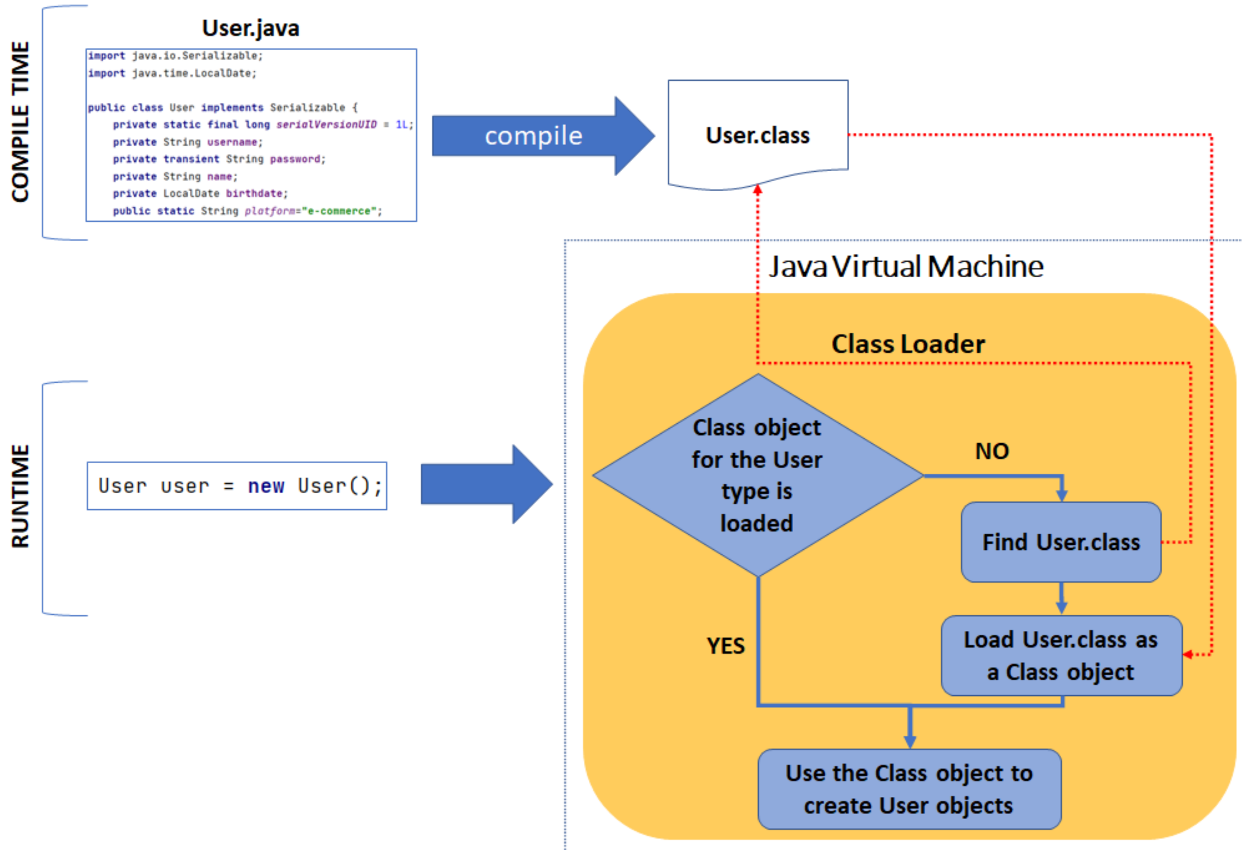
## Metadata includes information about

- The class itself, like package and superclass of the class
- The interfaces implemented by the class
- Details of the constructors, fields, and methods defined by the class

# Reflection

- Runtime type information
  - Discover and use while a program is running
  - Type information is represented at runtime as a **Class** object

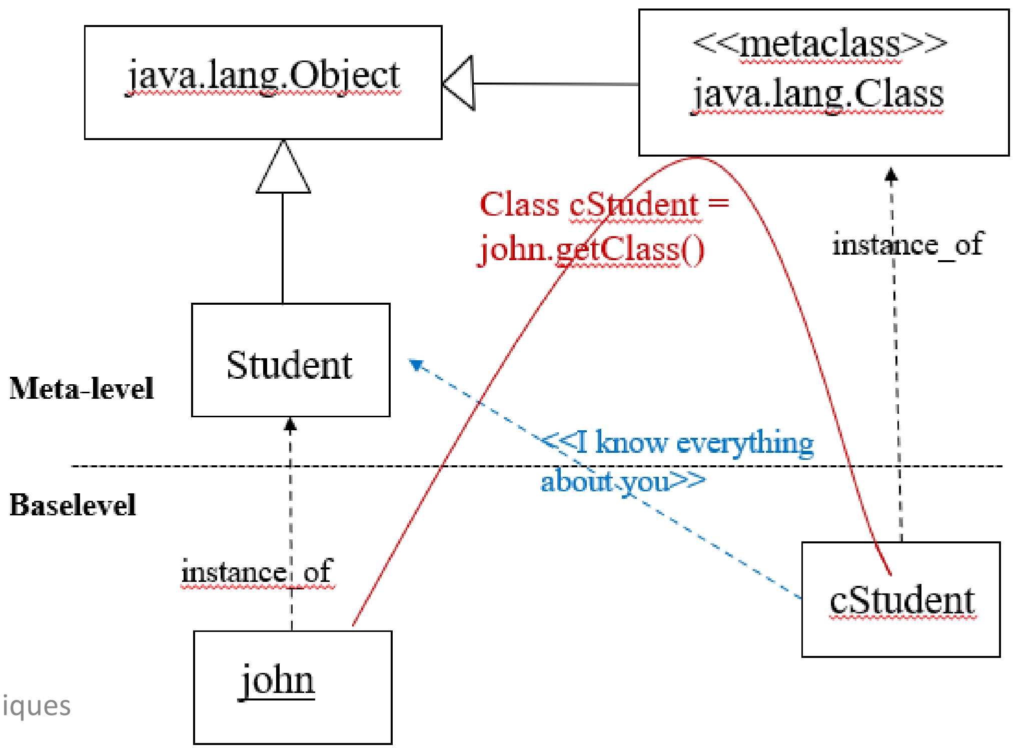
You must first get a reference to the appropriate Class object



# Reflection

- **Java.lang.Class** - Stores metadata for the class itself
  - Package and superclass, interfaces implemented
  - Details of the constructors, fields, and methods defined by the class

**Meta-level and base level**



# Reflection

- **Java.lang.Class** - building Class objects

```
Class c = “Alpha".getClass();

Point p = new Point (2.1, 3.2);
Class cp = p.getClass();

Set<String> s = new HashSet<>();
Class c = s.getClass();
```

**Getting a Class object**

(1) Use method **getClass()** of the class **Object** if you have a reference to an object

(2) Use the static method **forName()** of the class **Class** when the type and name is available

```
package foo;
public class Test {
  public Test () {
    System.out.println(“Hello Test");
  }

  public static void main(String[] args)
                                  throws Exception {
    Class cls = Class.forName(“foo.Test");
    Test tst = (Test) cls.newInstance();
    }
}
```

# Reflection

- **Java.lang.Class** - other metadata classes

Metadata for a class

Metadata for
- constructor,
- fields,
- methods,
- etc.

```
class Class {
  Constructor[] getConstructors();
  Field getDeclaredField(String name);
  Field[] getDeclaredFields();
  Method[] getDeclaredMethods();
  ...
}

…

class Field {
  Class getType();
  ...
}
class Method {
  Class[] getParameterTypes();
  Class getReturnType();
  ...
}
```

# Reflection

- **Reflection with methods**
  - Retrieve methods example

***Methods from the class Class***

*Method getMethod(String name, Class<?>... parameterTypes)*
*Method[] getMethods()*
*Method getDeclaredMethod(String name, Class<?>... parameterTypes)*
*Method[] getDeclaredMethods()*

```
public class Vector ... {
  public synchronized boolean addAll (Collection c) ...
  public synchronized void copyInto (Object[] anArray) ...
  public synchronized Object get (int index) ...
}

Querying class Vector for its method get:
Method m = Vector.class.getMethod("get", new Class[] {int.class});

Querying the Vector class for its addAll method
Method m = Vector.class.getMethod("addAll", new Class[] {Collection.class});

Querying the Vector class for its copyInto method
Method m = Vector.class.getMethod("copyInto", new Class[]{Object[].class});
```

# Reflection

> ***Methods from the class Method***
> *Class getDeclaringClass()*
> *Class[] getExceptionTypes()*
> *int getModifiers()*
> *String getName()*
> *Class[] getParameterTypes()*
> *Class getReturnType()*
> *Object invoke(Object obj, Object … args)*

- **Reflection with methods**
  - Retrieve all methods
  - Retrieve a specific method if you know the details
  - If you don't know the parameters, you can obtain them

```
Class classObject = ...//obtain class object
    Method[] methods = classObject.getMethods();
    classObject.getMethod(String name, Class[] parameterTypes)

    Class[] parameterTypes = method.getParameterTypes();
    Class returnType = method.getReturnType();
```

  - Call the method

```
aMethod.invoke(Object target, Object[] parameters)
```

```
Method m = cls.getMethod("doWork", new Class[]{String.class, String.class});

Object result= m.invoke(obj, new Object[]{"x","y"});
```

# Reflection

- **Reflection with methods -** Dynamic invocation
  - Call a method on an object at runtime without specifying which method at compile time
  - Example
    - **p** is a variable of type **Property** (e.g., mass, height, length, etc.)
    - Object[] is an array of arguments passed as parameters
    - If **setProperty** is static method of class **o,** the first parameter is ignored (**null**)
    - The return value of invocation is Object

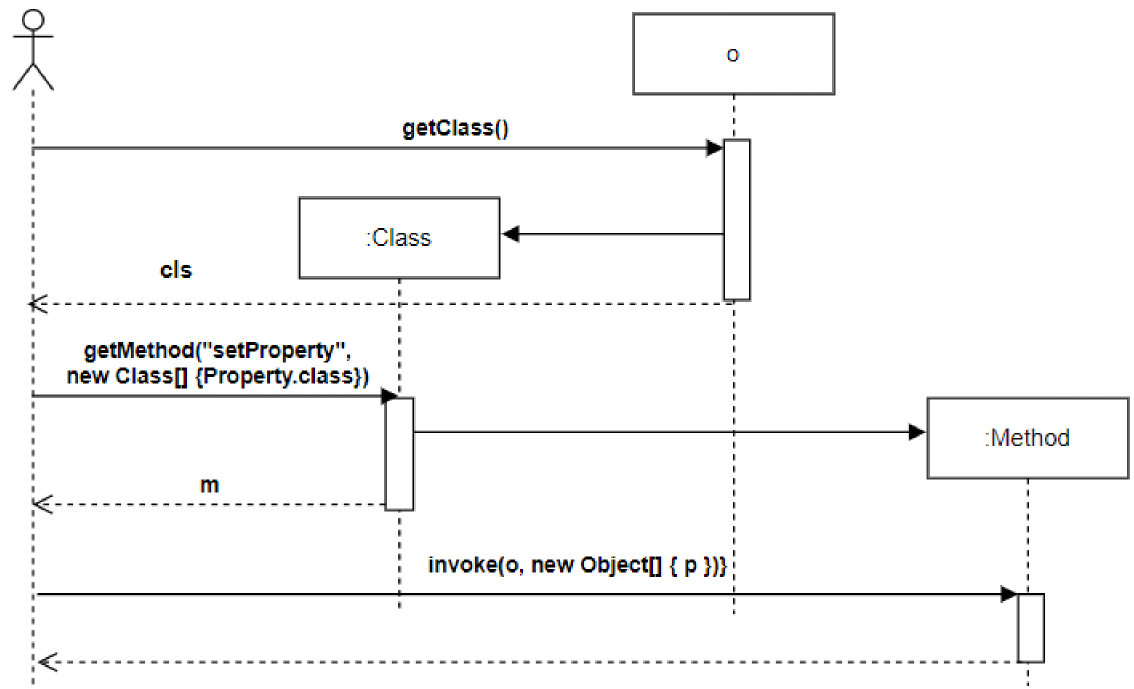
```
public static void setObjectProperty (Object o,  Property p) {
  Class cls = o.getClass();
  try {
   Method m = cls.getMethod("setProperty", new Class[] {Property.class});
    m.invoke (o, new Object[] { p } );
  }
   catch { Exceptions … }
}
```

# Reflection

- **Reflection with methods**

**Dynamic method invocation**

- Primitive types used as parameters are wrapped before calling (e.g., int is wrapped to Integer)
- Return type is wrapped before effective returning



# Reflection

- **Reflection with constructors**
  - Class methods for constructor introspection
  - Class objects returned by forName used to specify a parameter list
  - newInstance method of class Class
    - Builds an object of the target class' object
    - calls the default constructor

```
cls.getConstructor(new Class[] {String.class, String.class})

Class.forName("com.mysql.jdbc.Driver").newInstance();
```

- Query for a public constructor that takes two String objects as parameters
  - NoSuchMethodException - if there is no constructor for the parameter list specified

# Reflection

- **Reflection with constructors**
  - java.lang.Reflect.Constructor
    - Java Reflection API defines a metaclass for dealing with constructors
    - represents Java constructors
  - Interface to Constructor is like the interface to Method, except it supports a newInstance method instead of *invoke*
  - The reflective methods of Constructor class
    - *Constructor.newInstance() vs Class.newInstance()*

```
Constructor constructor = aObj.getClass().getConstructor(String.class);

 constructor.newInstance("arg constr");
```

# Reflection

- **Reflection with fields**
  - If **field** refers to a field object of the object **obj**, its value is accessed as
    - **`Object value = field.get(obj);`**
  - If the field type is primitive - Java wraps the value
    - `getBoolean, etc.`
  - Sets the value of the field
    - **`field.set(obj, value);`**
  - Specify a field that does not exist
    - NoSuchFieldException
  - Querying for fields can be disabled in the Java security manager
    - SecurityException
  - The return type of the methods is java.lang,reflect.Field
    - Information about field's name, declaring class, and modifiers
  - Useful for deserialization

***Class methods for fields introspection***

```
Field getField(String name)
Field[] getFields()
Field getDeclaredField(String name)
Field[] getDeclaredFIelds()
```

# Reflection

*__Methods defined in the Field class__*
*Class getType()*
*Class getDeclaringClass()*
*String getName()*
*int getModifiers()*
*Object get (Object obj )*
*boolean getBoolean (Object obj ) …*
*void set (Object obj, Object value )*
*void setBoolean ( Object obj, boolean value )*
*…*

- **Reflection with fields and modifiers**

```
// Getting field values
Object value = field.get(obj);

// Setting field values
field.set(obj, value);
```

```
if (!Modifier.isPublic(field.getModifiers()))
{
  field.setAccessible(true);
}
Object value = field.get(obj);
```

**Disables all JVM runtime access checks to field**

```
public static Field[] getInstanceVariables(Class cls) {
   List accum = new LinkedList();
   while (cls != null) {
      Field[] fields = cls.getDeclaredFields();
      for (int i=0; i<fields.length; i++) {
         if (!Modifier.isStatic(fields[i].getModifiers())) accum.add(fields[i]);
      }
      cls = cls.getSuperclass();
   }
   Field[] retvalue = new Field[accum.size()];
   return (Field[]) accum.toArray(retvalue);
}
```

# Reflection

**Methods defined in the Array class**

*Object newInstance(Class<?> componentType, int length)*
*newInstance(Class<?> componentType, int... dimensions)*
*int getLength(Object array)*
*boolean getBoolean ( Object array, int index)*
*void set(Object array, int index, Object value)*
*setBoolean(Object array, int index, boolean z)*
*Object get (Object array, int index)*

- **Reflection on arrays**
  - Class for performing reflective operations on all array objects
    - `java.lang.reflect.Array`
  - The length of an array
    - **`int length = Array.getLength(obj);`**
    - Assume obj refers to an array
  - Reflective access on the ith element of the array.
    - **`Array.get(obj, i)`**
    - get wraps the accessed value in its corresponding wrapper

```
// getting the length of an array
int length = Array.getLength(obj);

// get the ith  element of an array
Array.get(obj, i)
```

# Applications of Reflection

- **Junit**
  - Test methods are identified by an annotation - reflection is used to assign behavior to appropriate annotation at run time
  - After finding those methods
    - using @Test, @BeforeTest, @AfterTest, they are invoked using reflection
  - Naming conventions of methods are used to infer semantics
- **Auto-completion in a text editor**
  - Java editors and IDEs provide auto-completion.
  - The pop-up menu is populated by using Java reflection

# Applications of Reflection

- **Spring**
  - Uses reflection to create an object for each bean
  - The object's type is specified by the class attribute

```
<bean id="someID" class = "com… .DomainClass"
     <property name="someField" value="someValue" />
</bean>
```

  - When Spring processes this <bean> element will use to instantiate the corresponding class object

```
Class.forName("com.. .DomainClass")
```

  - After an object is constructed, each property is examined

```
obj.setXXX(value)
```

  - By default, the object is created with its default constructor