Lesson 3.8
Internal classes and Reflection

Software Analysis and Design

2nd Year, Computer Science

Universidad Autónoma de Madrid

Lesson 3.8 Anonymous classes & reflection (basics)

Software Analysis and Design

2nd Year, Computer Science

Universidad Autónoma de Madrid

Index

- **Anonymous classes**
- Reflection

Local anonymous classes

- A class that is declared inside a block of code

- An anonymous class is a local class with no name
  - declared an instantiated on the spot
  - defined over a base class, or an interface

- Can access:
  - the members of the external class
  - effectively final local variables of the container block of code

Anonymous classes

```
public class SimpleWindow {

  public static void main(String[] args) {
      // create window
      JFrame window = new JFrame("My GUI");

      // …
      JButton button   = new JButton(“Click me");
      final JTextField field = new JTextField(5);

      // bind actions to components
      button.addActionListener(
        new ActionListener() {
          public void actionPerformed(ActionEvent e) {
            JOptionPane.showMessageDialog(null, field.getText());
          }
        }
      );
      …
      }
}
```

Anonymous classes & Enumerations

- It is possible to declare abstract methods in an *enum*, and implement them in every enum object
- For that, we use anonymous classes

```
enum LogicalGate{
  AND {
    @Override Boolean calculate(Boolean b1, Boolean b2) {
      return b1 && b2;
    }
  }, OR {
    @Override Boolean calculate(Boolean b1, Boolean b2) {
      return b1 || b2;
    }
  };

  abstract Boolean calculate(Boolean b1, Boolean b2);
}
```

Index

- Internal classes
- **Reflection**

Reflection

- Inspect and change the behaviour of a program while it is running

- Very powerful technique
  - We can inspect the type of an object, access its attributes and methods, etc
  - We can create objects given a String with the class name
  - We can perform actions that otherwise would be illegal (e.g., access to private attributes or methods)

- Use caution
  - Less performance
  - Security restrictions (e.g., not possible for Applets)
  - Exposing internal members (e.g., private)

`instanceof` operator

- Infix binary operator

- Takes as parameters:
  - a reference, and
  - a class, interface or enum

- Returns if the object type at runtime is compatible with the type

- Its use normally signals a bad code design

Example

```
class A {}
class B extends A {}

public class Reflection1 {
   public static void main(String[] args) {
     A a = new B();
     if (a instanceof B)
        System.out.println(“Type B");
     else System.out.println(“Type A");
   }
}
```

Output: Type B

Example of BAD design

```
public abstract class Booking{
  protected String code;
  //….
  public String getCode() { /*…*/ }
}
```

```
public class HotelBooking extends Booking
{ //…}
```

```
public class FlightBooking extends Booking
{ //…}
```

```
public class BookingManager {
   private List<Booking> bookings = new ArrayList<>();
   public boolean cancel(String code) {
     Booking r = this.getBooking(code);
     if (r instanceof HotelBooking) { /* cancel Hotel Booking*/ }
     else if (r instanceof FlightBooking) { /* cancel Flight Booking */ }
     else if (r instanceof TravelBooking) { /* cancel Travel Booking*/ }
     //…
    return false;
   }
   private Booking getBooking(String code) { /*…*/ }
}
```

This design is BETTER

```
public abstract class Booking{
  protected String code;
  //….
  public String getCode() { /*…*/ }
  public abstract boolean cancel();
}
```

```
public class HotelBooking extends Booking
{
    public boolean cancel() { /*…*/}
}
```

```
public class FlightBooking extends Booking
{
    public boolean cancel() { /*…*/}
}
```

```
public class BookingManager{
   private List<Booking> bookings = new ArrayList<>();
   public boolean cancel(String code) {
     Booking r = this.getBooking(code);
     if (r==null) return false;
     return r.cancel();
   }
   private Booking getBooking(String code) { /*…*/ }
}
```

Why is it better?

The `Class` Class

- An object that represents a class (interface or enum)

- `Class` lacks public constructor. Its objects are built by the Java virtual machine.

- Access `Class` objects through the `getClass()` method of Object

```
public class Reflection2 {
  public static void main(String[] args) {
     Class<?> class = “a string".getClass();
     System.out.println("class = "+class);
  }
}
```

```
Output: class = class java.lang.String
```

Example

```
import java.lang.reflect.Field;
class Course{
 private String name = "PADS";
 public Course()   {}
 public Course(String name)    { this.name = name; }
 @Override public String toString() { return “Course = "+this.name; }
}

public class Reflection3 {
  public static void main(String[] args) throws
    ClassNotFoundException, InstantiationException,
    IllegalAccessException, NoSuchFieldException, SecurityException,
    IllegalArgumentException, InvocationTargetException, NoSuchMethodException
  {
    Class<?> clas = Class.forName("reflection.Course");

    Object asig = clas.getDeclaredConstructor().newInstance();   // 0-param constr
    System.out.println(asig);
    Field fld = clas.getDeclaredField("name");
    fld.setAccessible(true);                     // We can bypass privacity…
    fld.set(asig, "ADS");      // but it may not be a good idea!!!
    System.out.println(asig);
  }
}
```