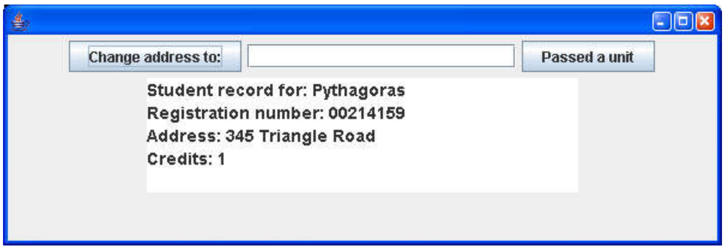
Objects and Classes (JFS Chap 9)

- Now (at last) we look at the aspects of Java that make it an "Object Oriented" programming language
- Topics:
  - A simple student record database (not in JFS)
  - The basis of objects
  - The concepts of "class" and "object"
  - An object-oriented version of the database program
  - Case study: JSlider

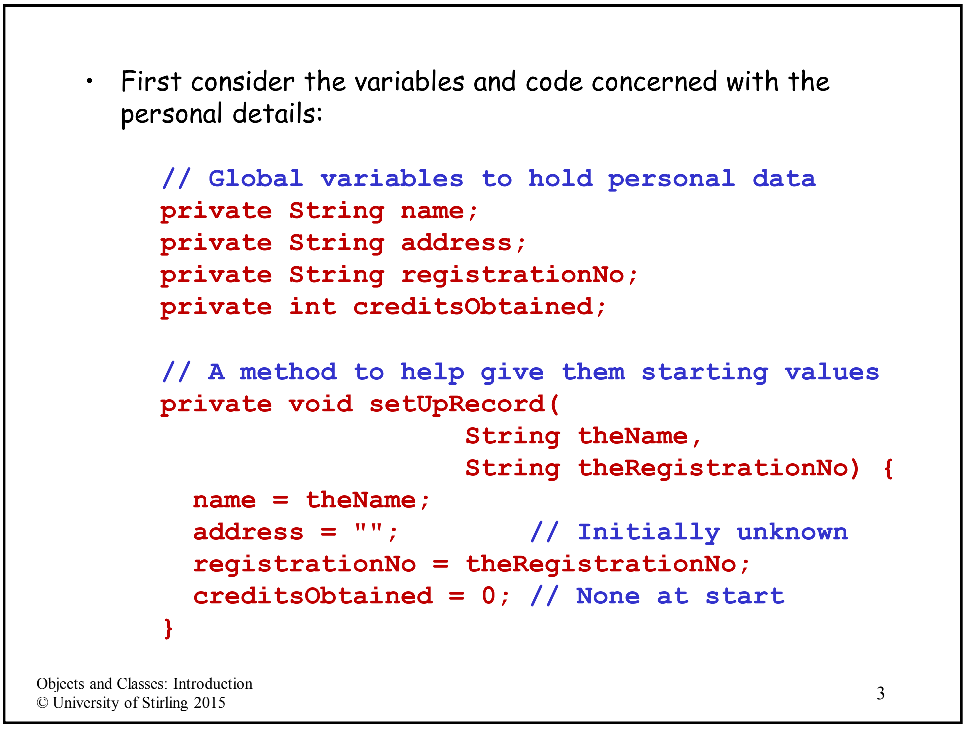
The Database program – simple version



Student record for: Pythagoras
Registration number: 00214159
Address: 345 Triangle Road
Credits: 1

- Pythagoras's details are displayed, and his address can be changed, and his credits increased
- The Java code is essentially straightforward

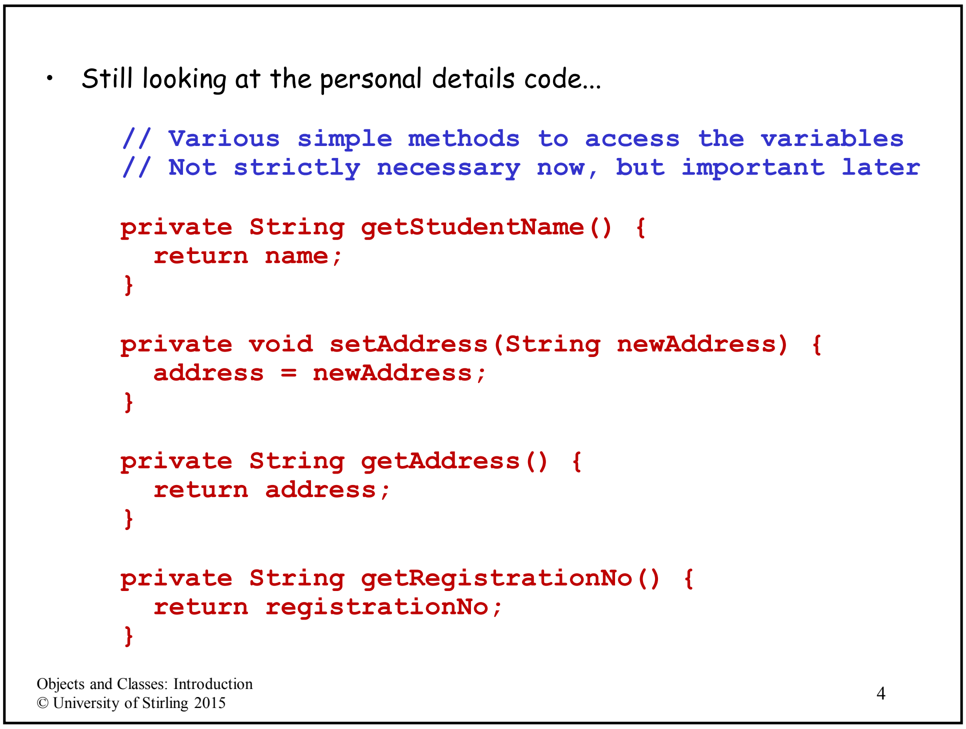
Demo
Database



- First consider the variables and code concerned with the personal details:

```
// Global variables to hold personal data
private String name;
private String address;
private String registrationNo;
private int creditsObtained;

// A method to help give them starting values
private void setUpRecord(
                   String theName,
                   String theRegistrationNo) {
  name = theName;
  address = "";         // Initially unknown
  registrationNo = theRegistrationNo;
  creditsObtained = 0; // None at start
}
```



- Still looking at the personal details code…

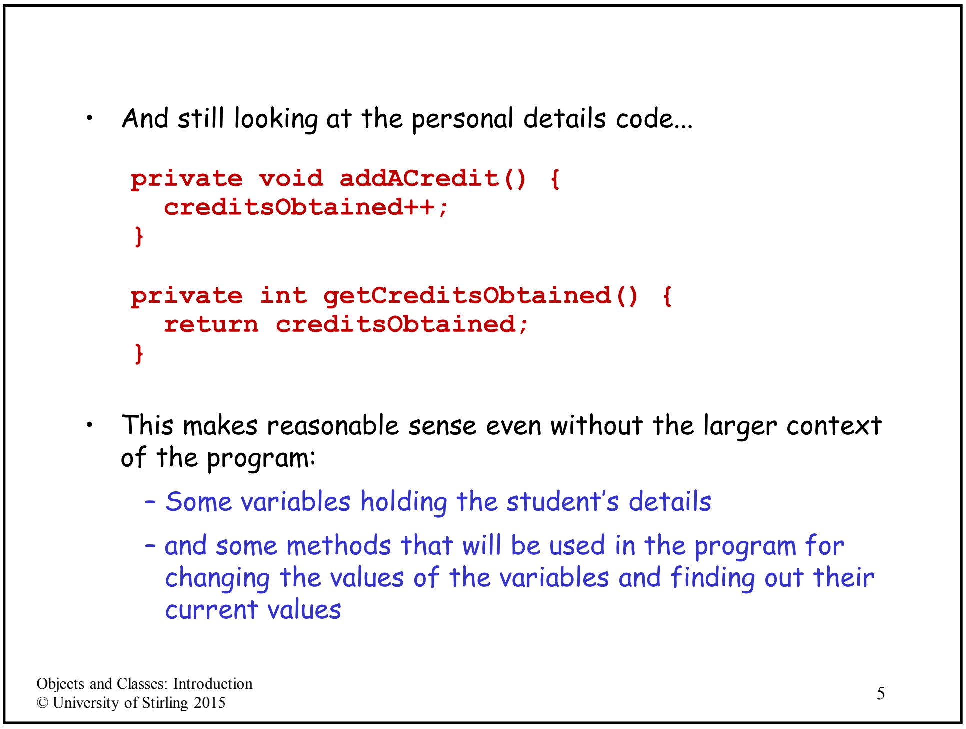
```
// Various simple methods to access the variables
// Not strictly necessary now, but important later

private String getStudentName() {
  return name;
}

private void setAddress(String newAddress) {
  address = newAddress;
}

private String getAddress() {
  return address;
}

private String getRegistrationNo() {
  return registrationNo;
}
```

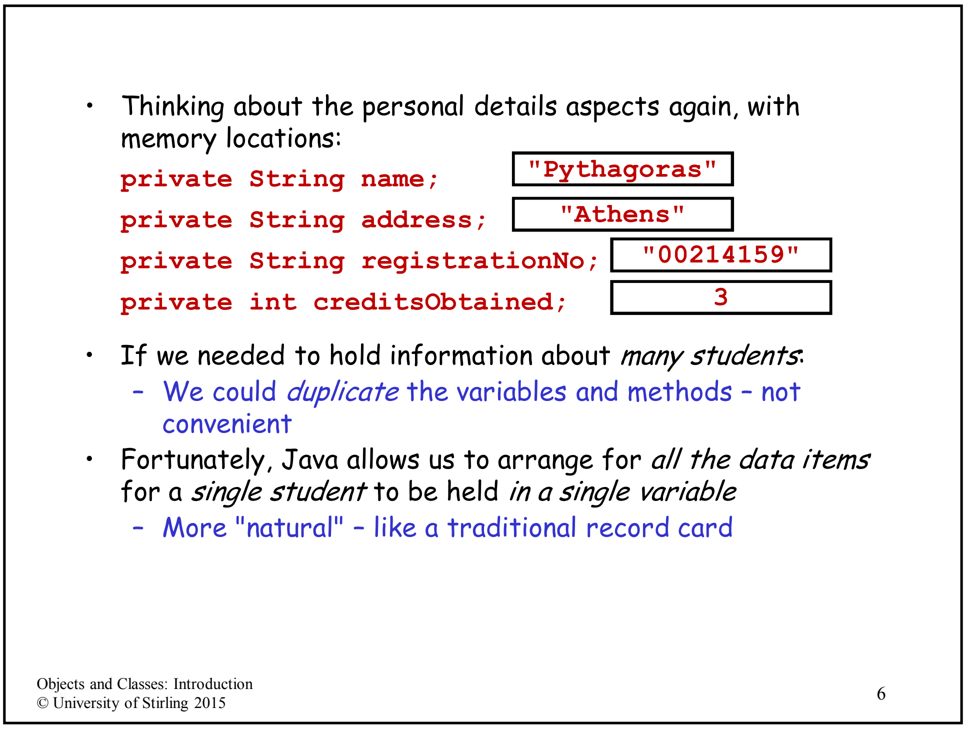


- And still looking at the personal details code...

```
private void addACredit() {
  creditsObtained++;
}

private int getCreditsObtained() {
  return creditsObtained;
}
```

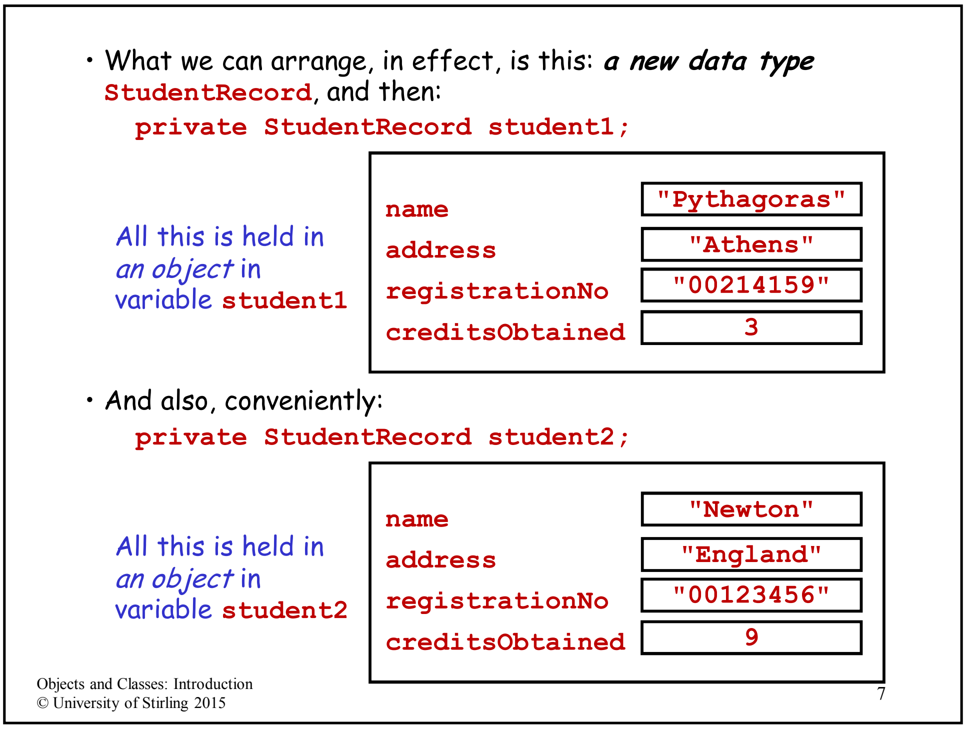
- This makes reasonable sense even without the larger context of the program:
  - Some variables holding the student's details
  - and some methods that will be used in the program for changing the values of the variables and finding out their current values



- Thinking about the personal details aspects again, with memory locations:

`private String name;`	"Pythagoras"
`private String address;`	"Athens"
`private String registrationNo;`	"00214159"
`private int creditsObtained;`	3

- If we needed to hold information about *many students*:
  - We could *duplicate* the variables and methods – not convenient
- Fortunately, Java allows us to arrange for *all the data items* for a *single student* to be held *in a single variable*
  - More "natural" – like a traditional record card



- What we can arrange, in effect, is this: ***a new data type*** `StudentRecord`, and then:

```
private StudentRecord student1;
```

All this is held in *an object* in variable `student1`

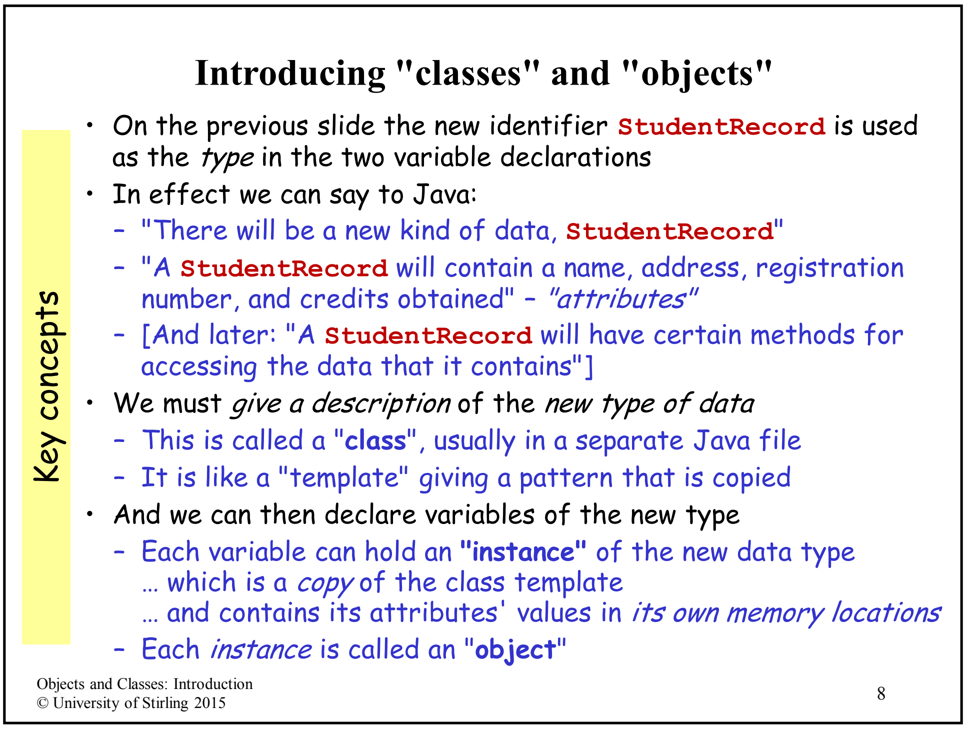
name	"Pythagoras"
address	"Athens"
registrationNo	"00214159"
creditsObtained	3

- And also, conveniently:

```
private StudentRecord student2;
```

All this is held in *an object* in variable `student2`

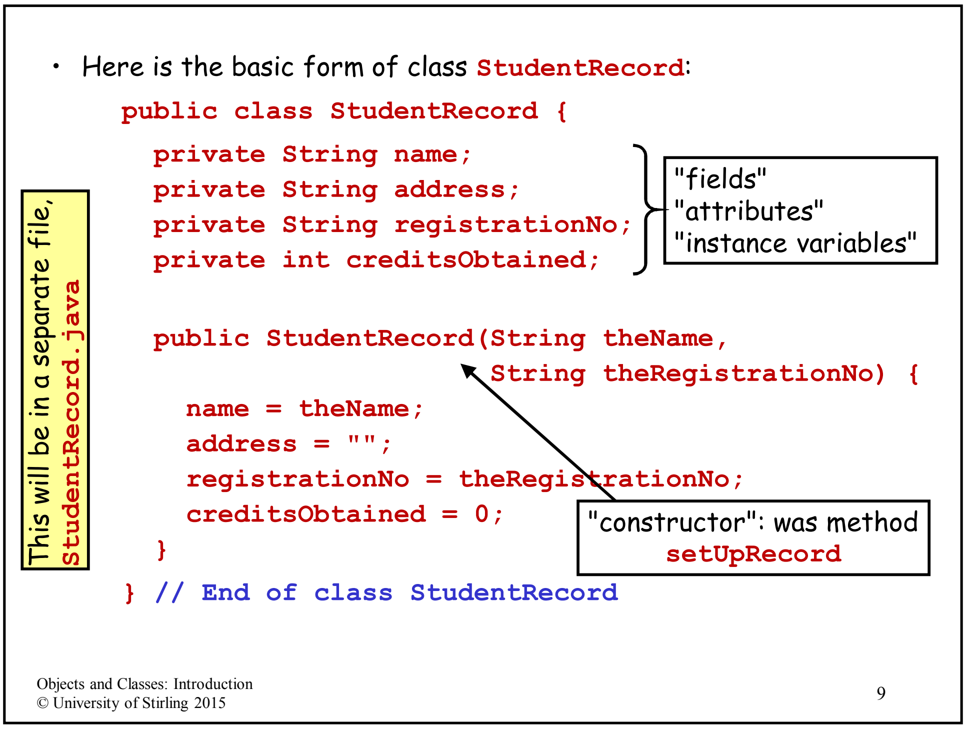
name	"Newton"
address	"England"
registrationNo	"00123456"
creditsObtained	9



Introducing "classes" and "objects"

Key concepts

- On the previous slide the new identifier `StudentRecord` is used as the *type* in the two variable declarations
- In effect we can say to Java:
  - "There will be a new kind of data, `StudentRecord`"
  - "A `StudentRecord` will contain a name, address, registration number, and credits obtained" – *"attributes"*
  - [And later: "A `StudentRecord` will have certain methods for accessing the data that it contains"]
- We must *give a description* of the *new type of data*
  - This is called a "**class**", usually in a separate Java file
  - It is like a "template" giving a pattern that is copied
- And we can then declare variables of the new type
  - Each variable can hold an **"instance"** of the new data type
    … which is a *copy* of the class template
    … and contains its attributes' values in *its own memory locations*
  - Each *instance* is called an "**object**"



- Here is the basic form of class `StudentRecord`:

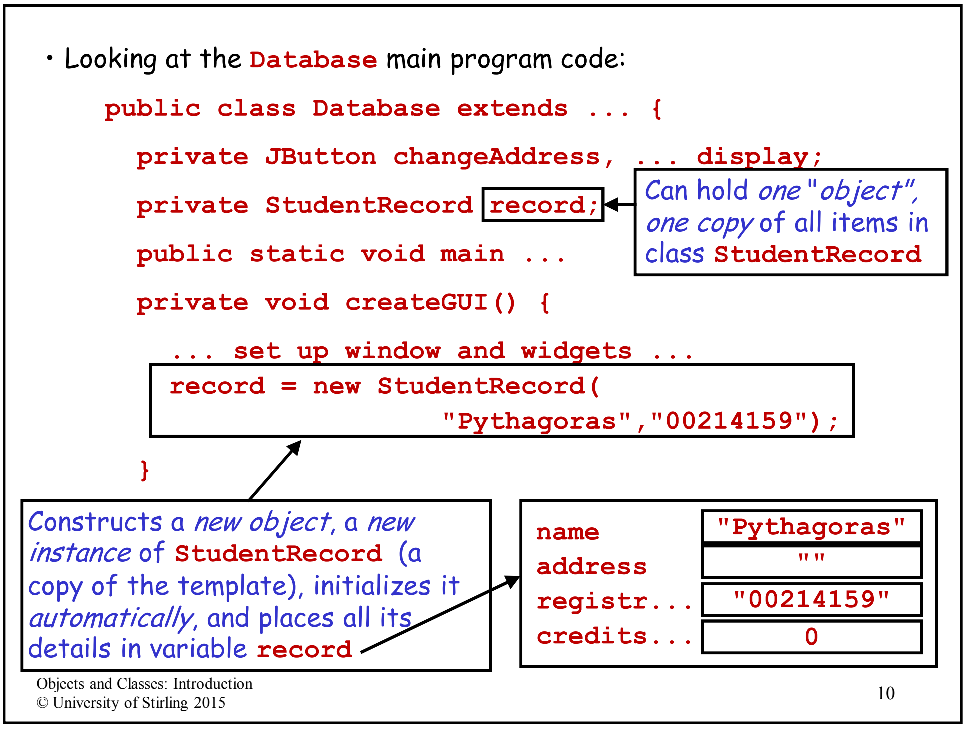
```
public class StudentRecord {
  private String name;
  private String address;
  private String registrationNo;
  private int creditsObtained;

  public StudentRecord(String theName,
                       String theRegistrationNo) {
    name = theName;
    address = "";
    registrationNo = theRegistrationNo;
    creditsObtained = 0;
  }
} // End of class StudentRecord
```

"fields"
"attributes"
"instance variables"

"constructor": was method `setUpRecord`

This will be in a separate file, `StudentRecord.java`



- Looking at the `Database` main program code:

```
public class Database extends ... {
  private JButton changeAddress, ... display;
  private StudentRecord record;
  public static void main ...
  private void createGUI() {
    ... set up window and widgets ...
    record = new StudentRecord(
                  "Pythagoras","00214159");
  }
```

Can hold *one* "*object*", *one copy* of all items in class `StudentRecord`

Constructs a *new object*, a *new instance* of `StudentRecord` (a copy of the template), initializes it *automatically*, and places all its details in variable `record`

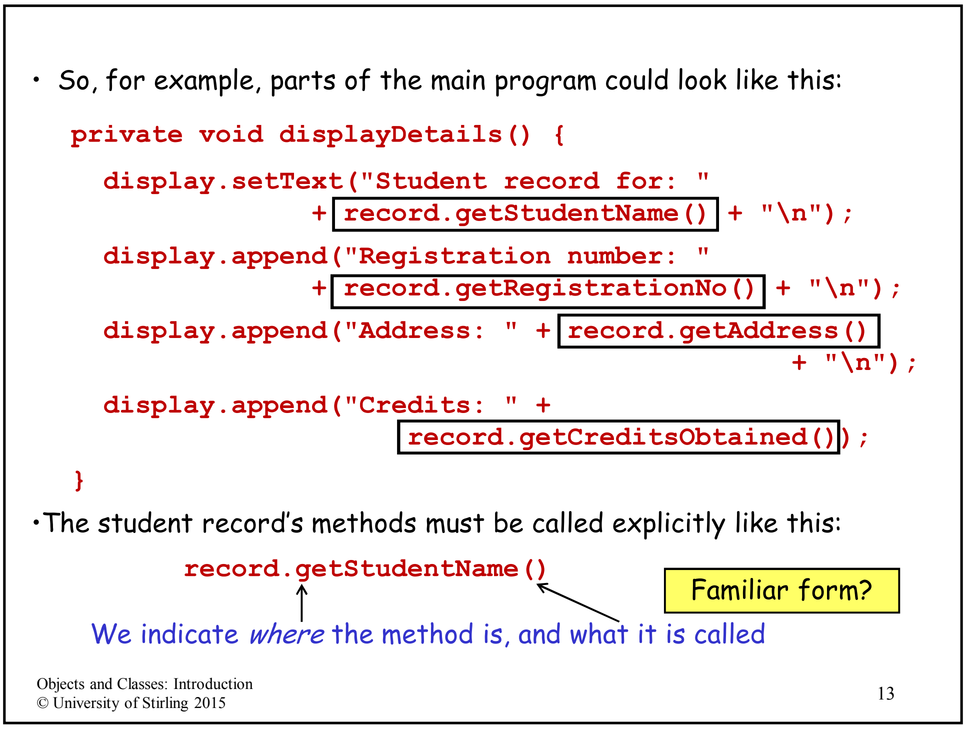
name	"Pythagoras"
address	""
registr...	"00214159"
credits...	0

- To allow the main program access to the student's data, *we need the get and set methods too*
- So finally class `StudentRecord` looks like this:

```
public class StudentRecord {
  ...name, address...
  public StudentRecord(...)...
  public String getStudentName() {
    return name;
  }
  public void setAddress(String newAddress) {
    address = newAddress;
  }
  ... etc
}
```

The `StudentRecord` class – key notes

- The class `StudentRecord` encapsulates a description of a student:
  - Characteristic data and what can be done with it
- The class `StudentRecord` *does not* `extend JFrame`
  - It doesn't have any user interaction facilities of its own, it's just about looking after student data
- The *methods* in the `StudentRecord` class are all `public`
  - They are intended to be called from the code in the main `Database` program class
  - There *could* be `private` methods too
- OTOH, the *global variables* in `StudentRecord` are `private`
  - The compiler *will not allow* them to be used directly by the code in the `Database` class
  - But we have *designed* `Database` to only call the methods!



- So, for example, parts of the main program could look like this:

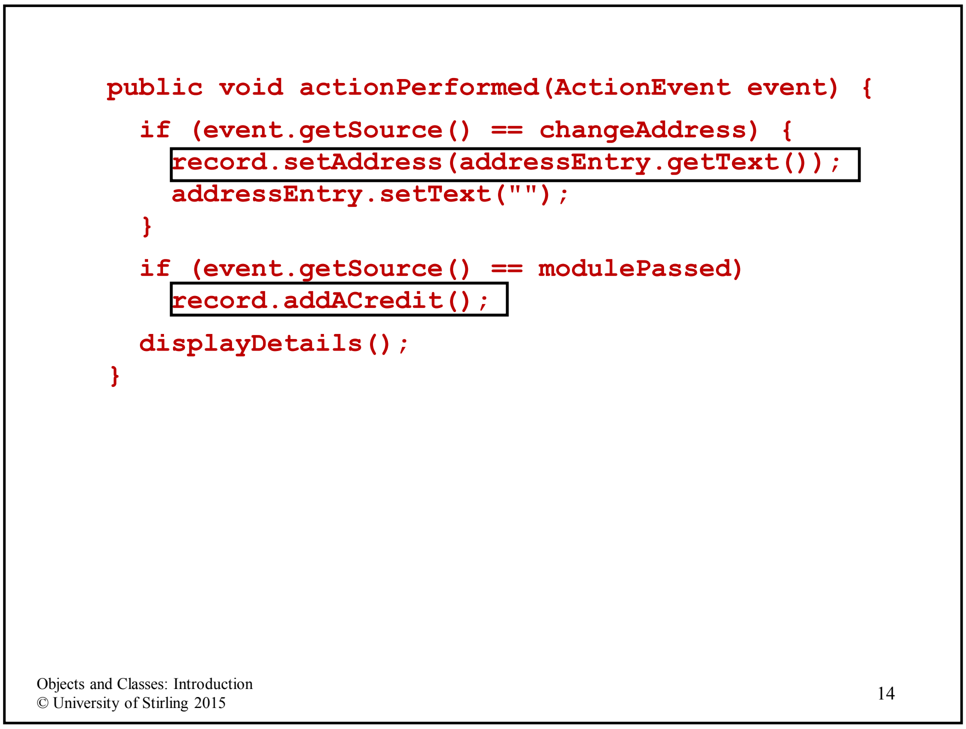
```
private void displayDetails() {
  display.setText("Student record for: "
               + record.getStudentName() + "\n");
  display.append("Registration number: "
               + record.getRegistrationNo() + "\n");
  display.append("Address: " + record.getAddress()
                                      + "\n");
  display.append("Credits: " +
                  record.getCreditsObtained());
}
```

- The student record's methods must be called explicitly like this:

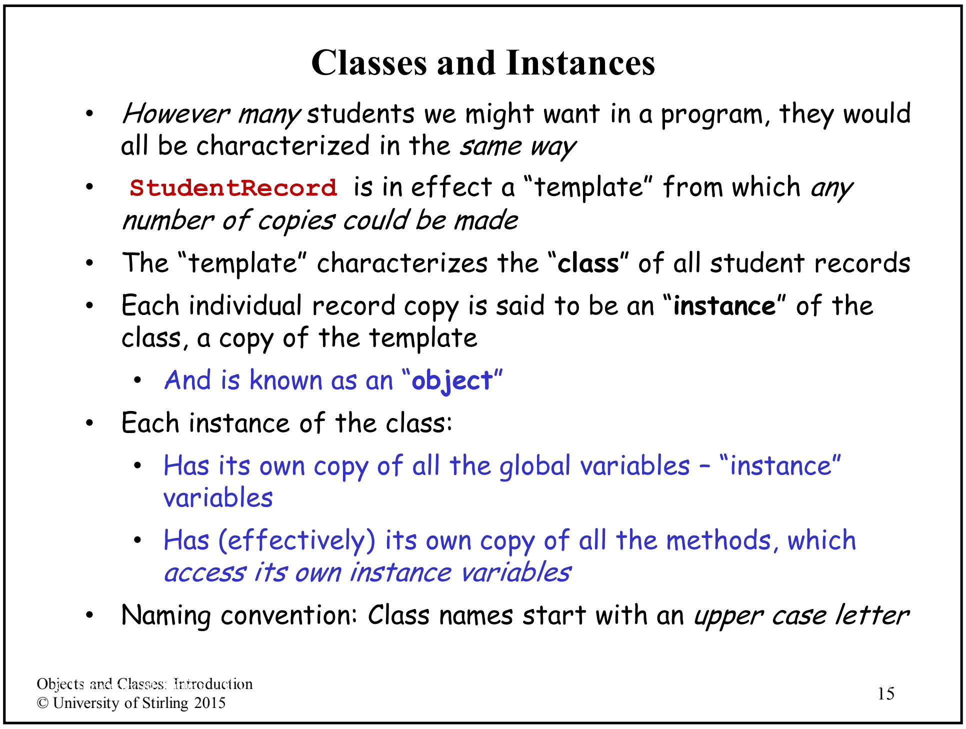
```
record.getStudentName()
```

Familiar form?

We indicate *where* the method is, and what it is called

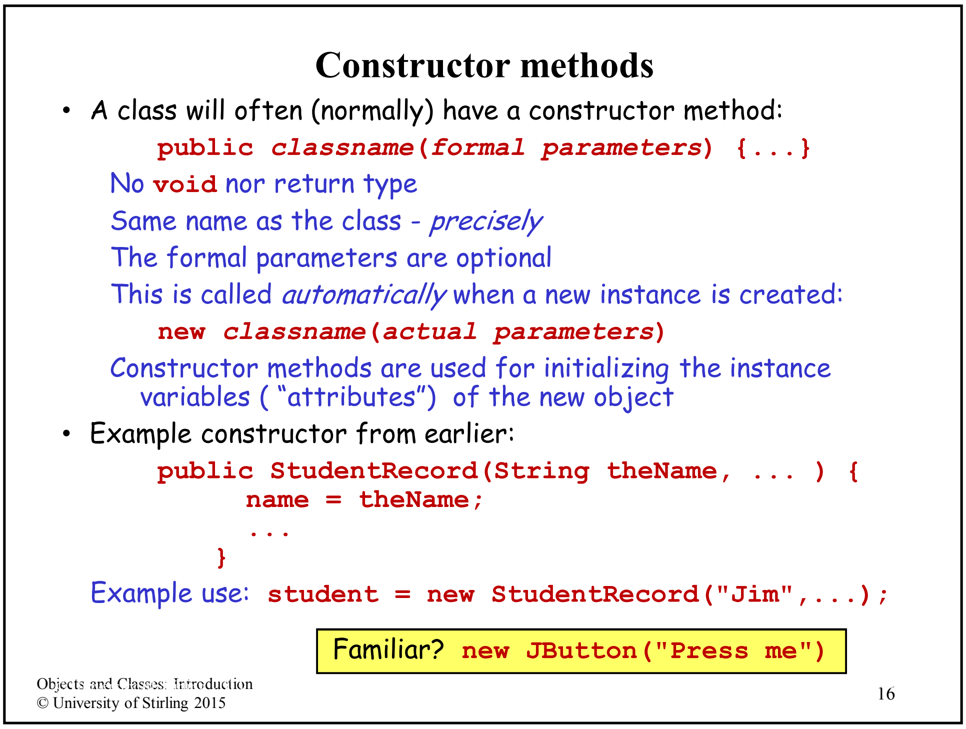


```
public void actionPerformed(ActionEvent event) {
  if (event.getSource() == changeAddress) {
    record.setAddress(addressEntry.getText());
    addressEntry.setText("");
  }
  if (event.getSource() == modulePassed)
    record.addACredit();
  displayDetails();
}
```



Classes and Instances

- *However many* students we might want in a program, they would all be characterized in the *same way*
- `StudentRecord` is in effect a "template" from which *any number of copies could be made*
- The "template" characterizes the "**class**" of all student records
- Each individual record copy is said to be an "**instance**" of the class, a copy of the template
  - And is known as an "**object**"
- Each instance of the class:
  - Has its own copy of all the global variables – "instance" variables
  - Has (effectively) its own copy of all the methods, which *access its own instance variables*
- Naming convention: Class names start with an *upper case letter*



Constructor methods

- A class will often (normally) have a constructor method:

```
    public classname(formal parameters) {...}
```

No `void` nor return type

Same name as the class - *precisely*

The formal parameters are optional

This is called *automatically* when a new instance is created:

```
    new classname(actual parameters)
```

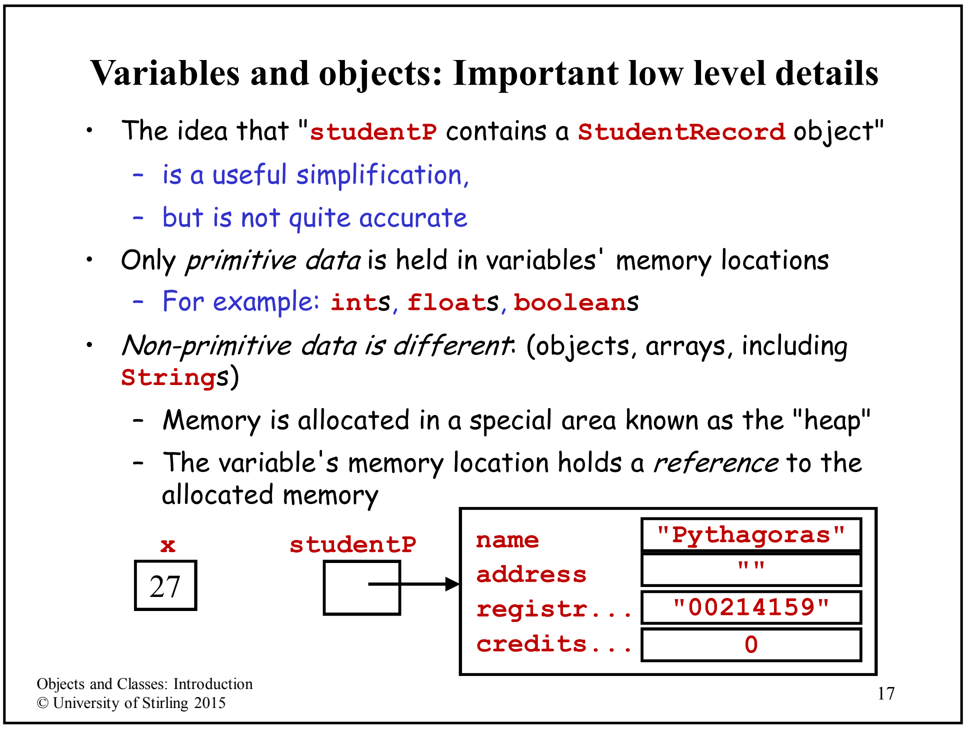
Constructor methods are used for initializing the instance variables ("attributes") of the new object

- Example constructor from earlier:

```
    public StudentRecord(String theName, ... ) {
           name = theName;
           ...
         }
```

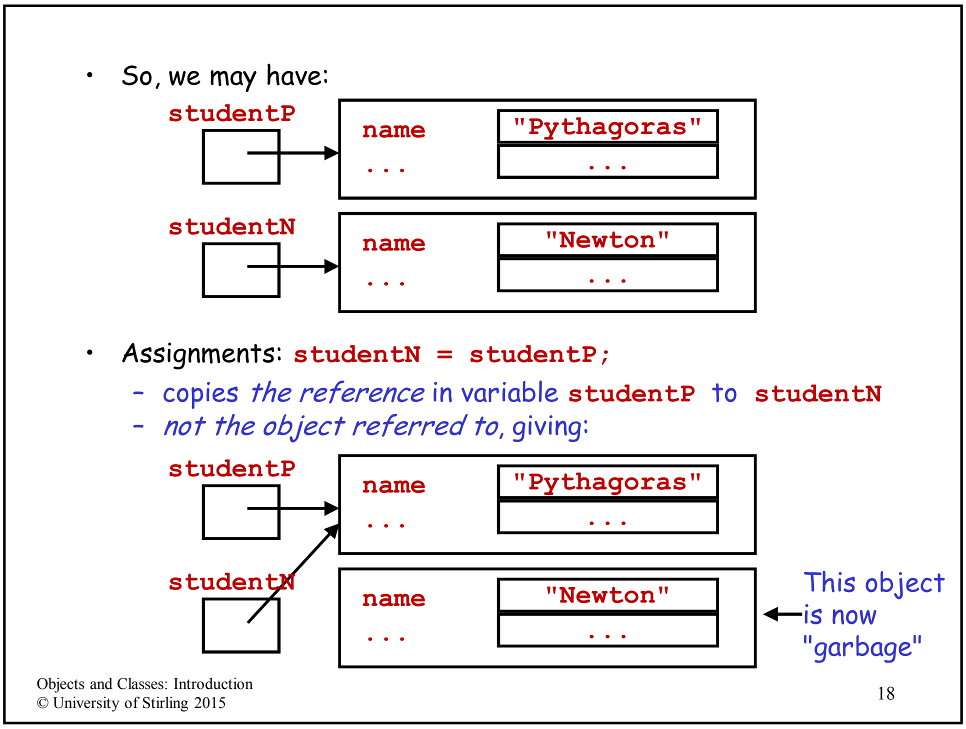
Example use: `student = new StudentRecord("Jim",...);`

Familiar? `new JButton("Press me")`



Variables and objects: Important low level details

- The idea that "`studentP` contains a `StudentRecord` object"
  - is a useful simplification,
  - but is not quite accurate
- Only *primitive data* is held in variables' memory locations
  - For example: `int`s, `float`s, `boolean`s
- *Non-primitive data is different*: (objects, arrays, including `String`s)
  - Memory is allocated in a special area known as the "heap"
  - The variable's memory location holds a *reference* to the allocated memory



- So, we may have:

- Assignments: `studentN = studentP;`
  - copies *the reference* in variable `studentP to studentN`
  - *not the object referred to*, giving:

Objects within objects

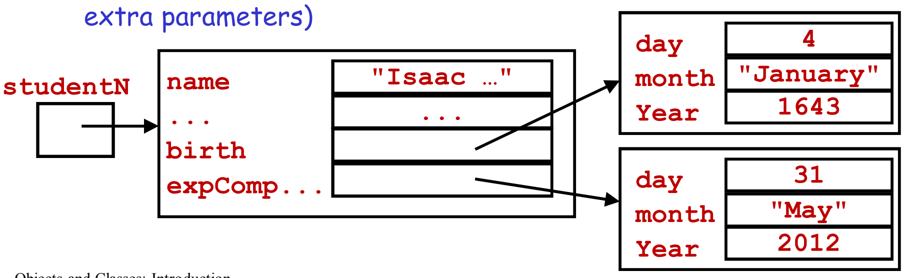
- The idea that any variable may contain an object is very general
  - In particular the attributes of any object may themselves contain objects
- For example: `Date`s within `StudentRecord`s

```
public class Date {
  private int day, year;
  private String month;
  ...
}
public class StudentRecord {
  ...
  private Date birth, expectedCompletion;
  ...
}
```

- And perhaps we then set up a new student record like this:

```
Date dateOfBirth = new Date(4, "January", 1643);
Date dateExpCompletion =
                   new Date(31, "May", 2012);
StudentRecord studentN =
    new StudentRecord("Isaac Newton", "00123456",
                 dateOfBirth, dateExpCompletion);
```

(The StudentRecord constructor has been extended with two extra parameters)



studentN

name "Isaac …"

... ...

birth

expComp...

day 4

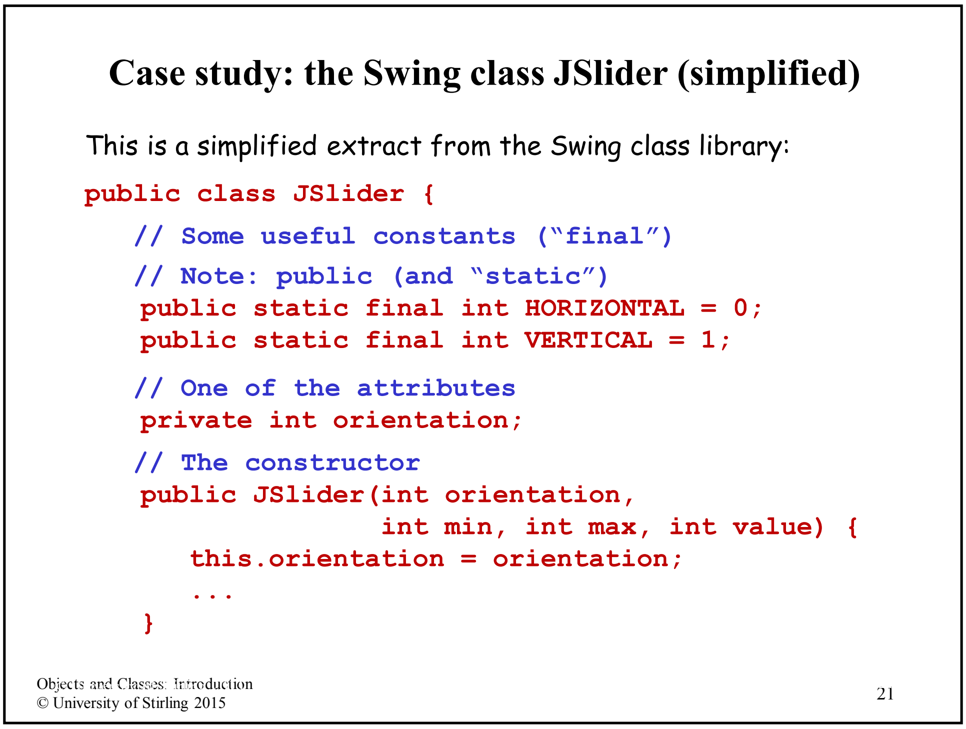
month "January"

Year 1643

day 31

month "May"

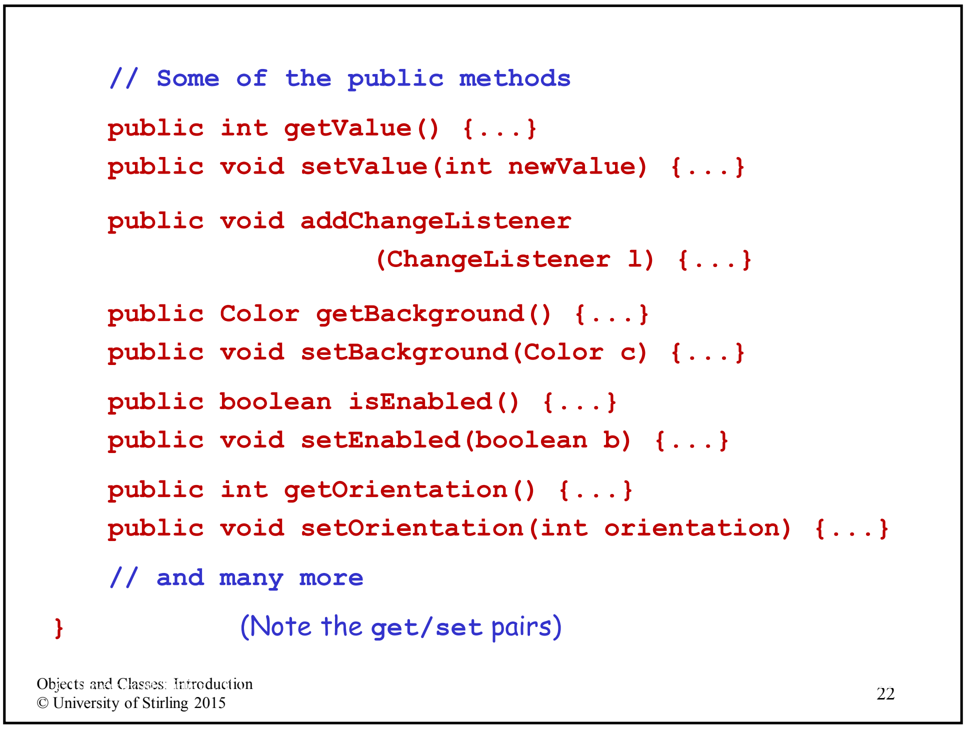
Year 2012



Case study: the Swing class JSlider (simplified)

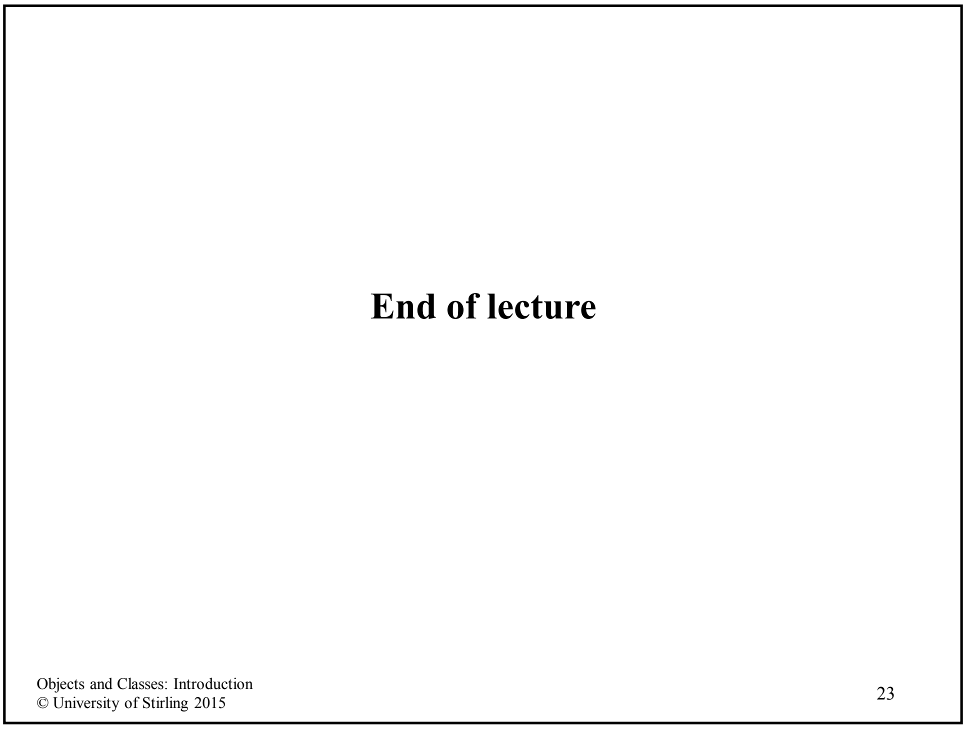
This is a simplified extract from the Swing class library:

```
public class JSlider {
   // Some useful constants (“final”)
   // Note: public (and “static”)
   public static final int HORIZONTAL = 0;
   public static final int VERTICAL = 1;
   // One of the attributes
   private int orientation;
   // The constructor
   public JSlider(int orientation,
                  int min, int max, int value) {
      this.orientation = orientation;
      ...
   }
```



```
   // Some of the public methods
   public int getValue() {...}
   public void setValue(int newValue) {...}
   public void addChangeListener
                    (ChangeListener l) {...}
   public Color getBackground() {...}
   public void setBackground(Color c) {...}
   public boolean isEnabled() {...}
   public void setEnabled(boolean b) {...}
   public int getOrientation() {...}
   public void setOrientation(int orientation) {...}
   // and many more
}
```

(Note the get/set pairs)



End of lecture