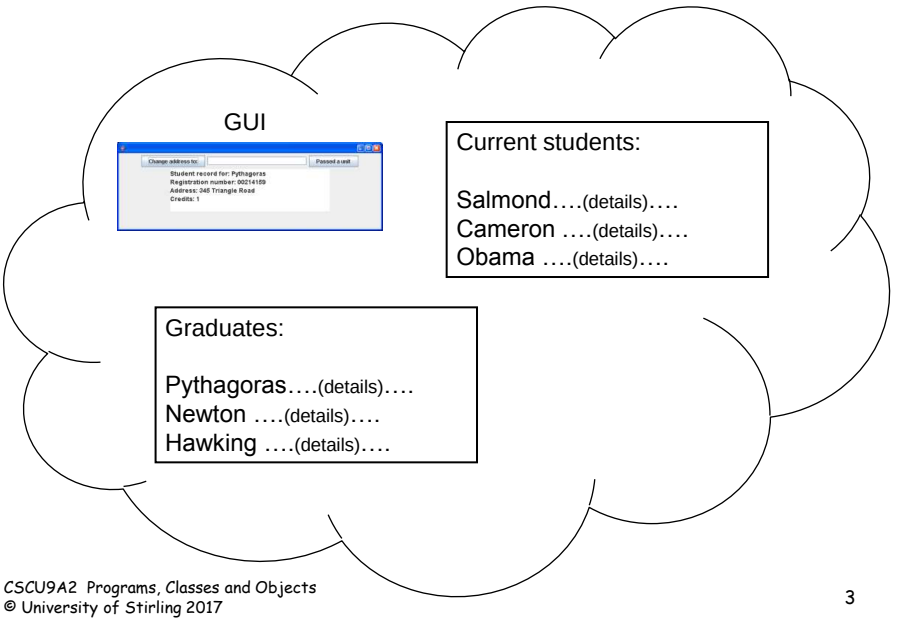
Programs, Classes and Objects

Horstmann, Chap 8, slide 6:
"Java programs are made of objects that interact with each other"

What does that mean?

- Suppose that we would like to design a software application for managing a *student record database*
- Some aspects that we will need to design and program:
  - There will be a graphical user interface (GUI)
  - There may be separate collections of student records
  - Each student record will store text items such as name, address, etc, and also some dates for first registration, expected end of studies, etc
  - Each date will be a day number, the name of the month, and a year number
  - The GUI accesses the collection, which accesses records...
- These different aspects correspond (more or less) with the *physical items* that would comprise a *manual* version of the records system
  - Think about what physical items...

Visualizing the system architecture



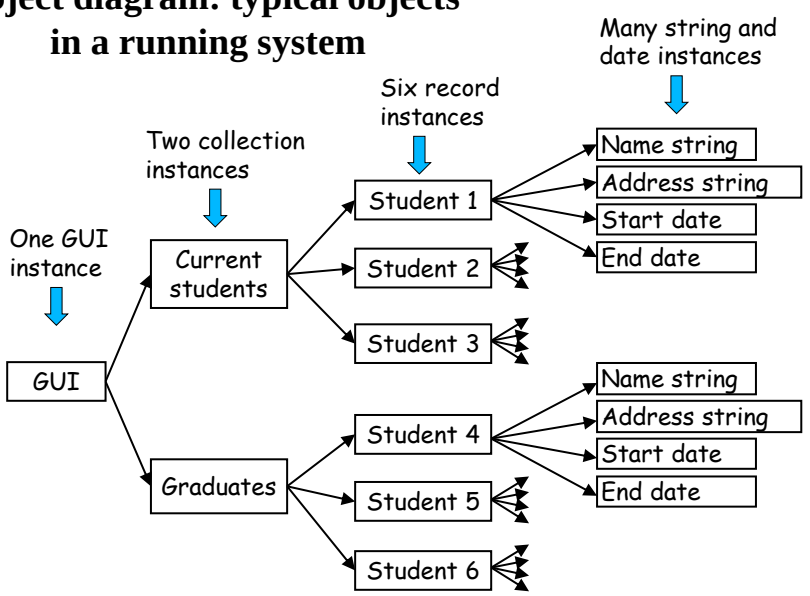
An engineering approach

- We *could* just build *one "monolithic" main program*:
  - One big class
  - With lots of variables: arrays, ints, Strings
  - And many methods to operate on them
- ***This would be very hard to build and maintain***
- But software engineers have discovered (invented) a better solution…
  - It took a couple of decades, 1960s – 1980s, of ideas and language design experiments to reach roughly what we have now
- ***With care, we can separate the program code associated with each separate aspect of the application into an independent, self-contained code component ("classes")***

- With care, we can separate the program code associated with each separate aspect of the application into an independent, self-contained code component ("classes")
  - Our software is then a structural "model" of (a manual version of) the system
  - Each "kind" of component is called a "class"
  - Each actual item in a running system is an "object" – an "instance" of a "class"
  - Objects communicate with each other as necessary (interaction)

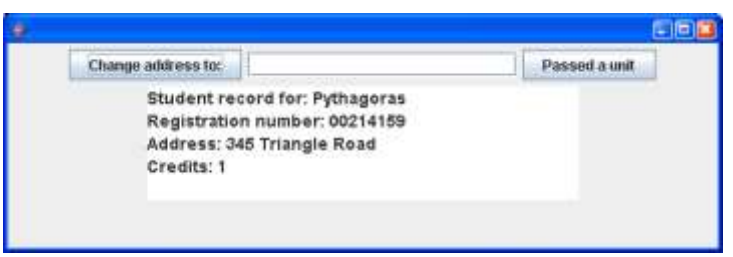
    For example: User makes query though the GUI, then the GUI "asks" a collection to find details for a particular student, then collection "asks" student record for its details (method calls)
- Advantages:
  - Structure is clearer, easier to navigate, build, maintain
  - Independent work on classes, independent testing...

Object diagram: typical objects in a running system



- Each object contains
  - Data relevant to the component
  - Methods to operate on the data
    – may also call methods in other objects: *communication*
- Example: The GUI
  - Data about the visual widgets: sizes, colours, positions, etc
  - Methods for handling user interactions with the widgets
    – which call collection methods
- Example: A collection of records
  - Data organizing the collection (e.g. array)
  - Methods to add, delete, find, update records
    – which call record methods
- And so on...
- "Object diagram" on previous slide – a typical running scenario

The origin of objects – A simple example record program



- Program administers the record for one student
- Pythagoras's details are displayed, and his address can be changed, and his credits increased
- The Java code is essentially straightforward

- 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 main 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;
private String address;
private String registrationNo;
private int creditsObtained;
```

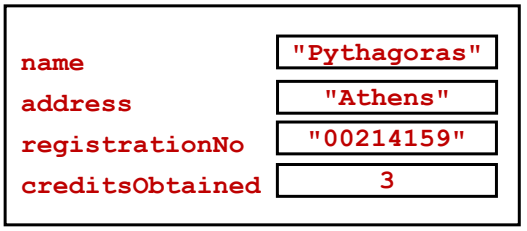
"Pythagoras"
"Athens"
"00214159"
3

- If we needed to hold information about *many students*:
  - We could *duplicate* the variables and methods
  - Inconvenient, not manageable
- 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*
  - The collection of data items is called an "object"

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

```
private StudentRecord student1;
```

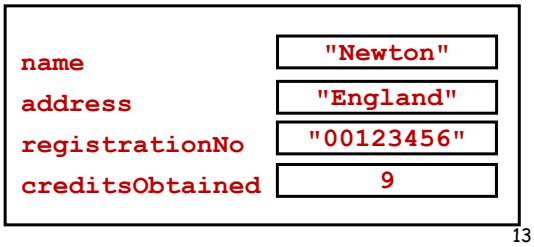
All this could be held in *an object* in variable `student1`



- And also, conveniently:

```
private StudentRecord student2;
```

All this could be held in *an object* in variable `student2`



Introducing "classes" and "objects"

Key concepts on one slide!

- 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
```

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

"fields"
"attributes"
"instance variables"

"constructor": was method `setUpRecord`

- This defines a *template* for a object instantiation

- In the main program:
  - Variables to hold instances of `StudentRecord` are declared like this:

```
private StudentRecord student1;
```

  - And instances are created like this:

```
student1 =
   new StudentRecord("Pythagoras","00214159");
```

- `... = new ...` means:
  - Allocate memory for a new object
  - Call its constructor, passing parameters, to initialize fields
  - Assign new object to the variable

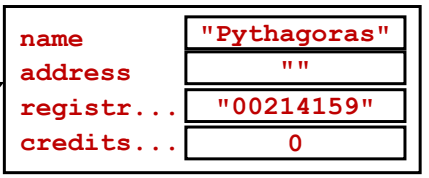
Familiar?
Jbuttons, etc

- The `Database` main program code will look like this:

```
public class Database extends ... {
  private JButton changeAddress, ... display;
  private StudentRecord record;
  public static void main ...
  private void setUpData() {
    ... set up any other data ...
    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`



- 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
}
```

Note:
`public` methods,
`private` data

- 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());
}
```

Note:
`record.` method-name`(...)` to call method inside an object


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

- The main program does *not contain* methods called `setAddress`, …
- So, the student record's methods must be called explicitly like this:

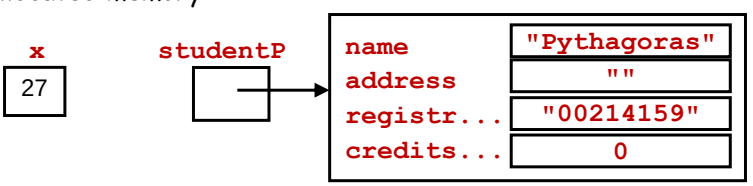
```
record.setAddress(...);
```

Familiar form?

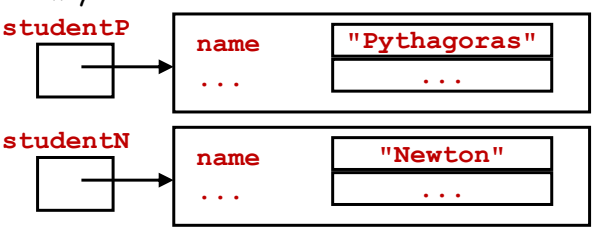
- We indicate *where* the method is, and what it is called
- The method is executed *within* its object, accessing the variables there

Variables and objects: Important low level details

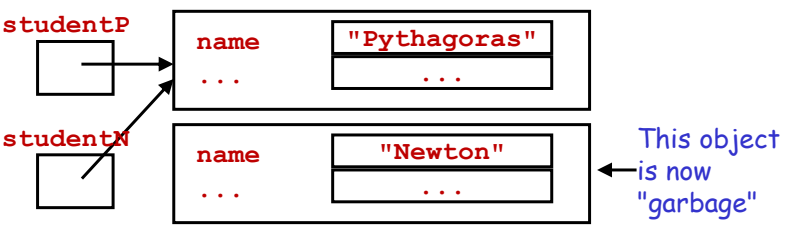
- The idea that "`studentP` contains a `StudentRecord` object"
  - is a useful simplification,
  - but is not quite accurate
- Only *primitive data* is held in variables' memory locations
  - For example: `int`s, `float`s, `boolean`s
- *Non-primitive data is different*: (objects, arrays, including `String`s)
  - Memory for `new` objects is allocated in 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

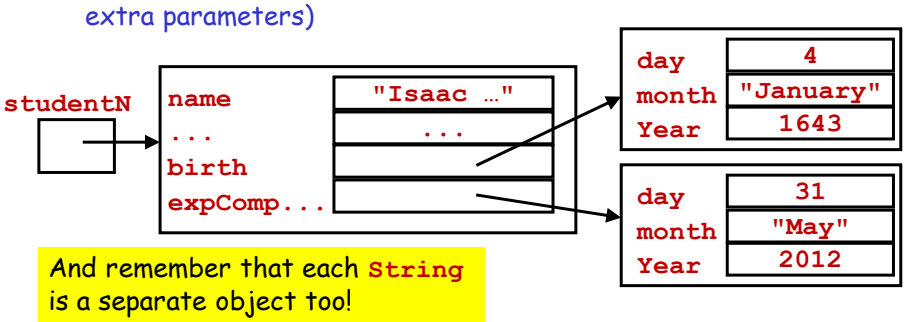
```
public class Date {
  private int day, year;
  private String month;
  ...
}
public class StudentRecord {
  ...
  private Date birth, expectedCompletion;
  ...
}
```

Two files

- 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)



And remember that each `String` is a separate object too!

End of section