University of HUDDERSFIELD

CFS2160: Software Design and Development

Lecture 10: Some OO Concepts

That we perhaps glossed over.

Tony Jenkins
A.Jenkins@hud.ac.uk

Object-Oriented?

Object-oriented?

In essence, OO programming is programming in terms of *objects*.

Objects have state (represented by their instance variables) and behaviours.

In general, objects represent things in the "real world".

Related objects form classes.

Object-oriented?

In essence, OO programming is programming in terms of *objects*.

Objects have state (represented by their instance variables) and behaviours.

In general, objects represent things

Related objects form classes.

In the mid-1980s there was a fashion for applying this concept to databases.
It never really caught on.

Object-oriented?

In essence, OO programming is programming in terms of *objects*.

Objects have state (represented by their instance variables) and behaviours.

In general, objects represent things

Related objects form classes.

However, you can think of a "class" as a table in a database. And an "object" as a single row in that table.

Benefits?

It is claimed that objects are a more natural way to write programs, as the view is closer to how we view the world.

It is also claimed that using OO allows us to build up libraries of classes that can be used in many applications.

Benefits?

It is claimed that objects are a more natural way to write programs, as the view is closer to how we view the world.

It is also claimed that using OO allows us to build up libraries of classes that can be used in many applications.

These are *claims*.

Languages

C is a purely procedural language.

Objects appeared in Smalltalk.

C++ added objects into C (along with a bunch of other stuff).

Python has objects, but can be used purely procedurally.

Java *can* be used procedurally, but only really makes sense when you realise *everything* is an object.

Did you say Python has Classes?

Indeed.

Here is a class, with a constructor.

```
class ClubMember:

  def __init__ (self, name, cash):
    self.name = name
    self.cash = cash
```

Did you say Python has Classes?

Indeed.

Here is a class, with a constructor.

And a method to return a string representation.

```
class ClubMember:

  def __init__ (self, name, cash):
    self.name = name
    self.cash = cash

  def __str__ (self):
    s = self.name + ' : ' + str (self.cash)
```

Did you say Python has Classes?

Indeed.

Here is a class, with a constructor.

And a method to return a string representation.

Now an object is created, and printed.

```
class ClubMember:

  def __init__ (self, name, cash):
    self.name = name
    self.cash = cash

  def __str__ (self):
    s = self.name + ' : ' + str (self.cash)

if __name__ == '__main__':

  gary = ClubMember ('Gary', 25)
  print (gary)
```

Did you say Python has Classes?

Indeed.

Here is a class, with a constructor.

And a method to return a string representation.

Now an object is created, and printed.

And here is its state.

```
class ClubMember:

  def __init__ (self, name, cash):
    self.name = name
    self.cash = cash

  def __str__ (self):
    s = self.name + ' : ' + str (self.cash)

if __name__ == '__main__':

  gary = ClubMember ('Gary', 25)
  print (gary)

  print (gary.name, gary.cash)
```

Did you say Python has Classes?

Indeed.

Here is a class, with a constructor.

And a method to return a string representation.

No

An

What, no getters?
Well, no.
It's not Pythonic, you see.

```
class ClubMember:

  def __init__ (self, name, cash):
    self.name = name
    self.cash = cash

  def __str__ (self):
    s = self.name + ' : ' + str (self.cash)

if __name__ == '__main__':

  gary = ClubMember ('Gary', 25)
  print (gary)

  print (gary.name, gary.cash)
```

Encapsulation

Encapsulation

Put simply, *encapsulation* is all about hiding the details of how a class works from users.

In terms of implementation, it means:

- Details of the implementation of the state are hidden within the class ("private").
- The state is manipulated via an interface ("public").

Encapsulation

Put simply, *encapsulation* is all about hiding the details of how a class works from users.

In terms of implementation, it means:

- Details of the implementatio within the class ("private").
- The state is manipulated via

In Java terms:
Instance variables are private.
Methods are public.

Encapsulation

Put simply, *encapsulation* is all about hiding the details of how a class works from users.

In terms of implementation, it means:

- Details of the implementatio within the class ("private").
- The state is manipulated via

Although methods that are only used inside the class are best made private.

Why Encapsulation?

It improves maintainability.

- The inner details of a class can be enhanced, as long as the external interface stays the same.

The integrity of the state is protected.

- Changes to state can be checked, and invalid changes rejected.

Why Encapsulation?

It improves maintainability.

- ➢ The inner details of a class can be enhanced, as long as the external interface stays the same.

The integrity of the state is protecte

- ➢ Changes to state can be chec
  rejected.

"Data Hiding" is a closely related term, which perhaps better describes what is going on.

Encapsulation in Action

Last week we used `ArrayList`.

We have no need to know *how* this collection works behind the scenes.

We just need to know *how* to use it.

As long as the interface stays the same, clever people can improve the implementation and all will be well.

Encapsulation in Action

Last week you used a class to represent club members.

You then wrote a class to represent the club itself; this class did not need to know how the first was implemented.

It just needed to know *how* to use it.

As long as the interface stays the same, you could improve the implementation and all will be well.

Example: Accessors and Mutators

Getters and Setters

Up to know, we have kind of automatically created accessors ("getters") and mutators ("setters") for every instance variable.

We have also seen how setters can include some simple logic, and how they can return a value to indicate success.

Getters and Setters

Up to know, we have kind of automatically created accessors ("getters") and mutators ("setters") for every instance variable.

But really we should only include those getters and setters that are needed.

Getters and Setters

Up to know, we have kind of automatically created accessors ("getters") and mutators ("setters") for every instance variable.

But really we should only include those getters and setters that are needed.

And we could have getters (and maybe even setters) that do not correspond to instance variables.

An Illustrative Example

Employees have a name and rating.

Ratings run from 0 to 10.

Ratings 0 to 2 are "Poor", 3 to 7 are "OK", 8 to 10 are "Excellent".

An Illustrative Example

Employees have a name and rating.

Ratings run from 0 to 10.

Ratings 0 to 2 are "Poor", 3 to 7 are "OK", 8 to 10 are "Excellent".

```
public class Employee {

  private String name;
  private int rating;
  private String ratingDesc;
```

An Illustrative Example

Employees have a name and rating.

Ratings run from 0 to 10.

Ratings 0 to 2 are "Poor", 3 to 7 are "OK", 8 to 10 are "Excellent".

No!

```
public class Employee {

  private String name;
  private int rating;
  private String ratingDesc;
```

An Illustrative Example

Employees have a name and rating.

Ratings run from 0 to 10.

Ratings 0 to 2 are "Poor", 3 to 7 are "OK", 8 to 10 are "Excellent".

The description cannot have a setter, but it can have a getter.

```
public class Employee {

  private String name;
  private int rating;
```

An Illustrative Example

Employees have a name and rating.

Ratings run from 0 to 10.

Ratings 0 to 2 are "Poor", 3 to 7 are "OK", 8 to 10 are "Excellent".

The description cannot have a setter, but it can have a getter.

```
public class Employee {

  private String name;
  private int rating;

.
.
.

  public String getRatingDescription () {
    if (this.rating <= 2) {
      return "Poor";
     }
     else if (this.rating <= 7) {
       return "OK";
     }
     else {
       return "Excellent";
     }
  }
```

An Illustrative Example

Employees have a name and rating.

Ratings run from 0 to 10.

Ratings 0 to 2 are "Poor", 3 to 7 are "OK", 8 to 10 are "Excellent".

Th
bu

Important
Any programmer using this class would have no idea that this String was not stored in the state.

```
public class Employee {

  private String name;
  private int rating;

.
.
.

  public String getRatingDescription () {
    if (this.rating <= 2) {
      return "Poor";
     }
     else if (this.rating <= 7) {
       return "OK";
     }
     else {
       return "Excellent";
     }
  }
```

Cohesion

Cohesion

Cohesion is a concept relating to what a class *does*.

A class should do *all* of *exactly one* thing.

If a class's responsibilities (functions) represent a meaningful unit, it has *high cohesion*, which is good.

Otherwise, *low cohesion* is bad.

Cohesion

Cohesion is a concept relating to what a class *does*.

A class should do *all* of *exactly one* thing.

If a class's responsibilities (functions) represent a meaningful unit, it has *high cohesion*, which is go

Otherwise, *low cohesion* is bad.

Quick Rule of Thumb
Describe what a class represents. If you use the word "and" it may well have low cohesion.

Cohesion is Good

Why is high cohesion good?

Simply, if a class implements exactly one meaningful thing it is likely to do it well.

Cohesion is Good

Why is high cohesion good?

An analogy:

- Washing machines wash clothes well.
- Tumble dryers dry clothes excellently.
- Washer-dryers can be an absolute nightmare.

Cohesion is Good

Why is high cohesion good?

Another analogy:

- A Hi-Fi system made of separate components usually produces better sound than an "all in one" music system.
- Each component in a Hi-Fi is very good at its single function.

An Illustrative Example

A well written class for Christmas Club members last week would implement all the behaviours of a member.

Nothing more, nothing less.

```
public class Member {

  private String name;
  private int balance;

  public Member (String name, int balance) {

  public String getName () …
  public void setName () …

  public boolean contribute (int amount) …
  public int getBalance () …
```

An Illustrative Example

A well written class for Christmas Club members last week would implement all the behaviours of a member.

Nothing more, nothing less.

It does **not** print messages, provide a dialogue to create a member, send emails to members about late payments, …

```
public class Member {

  private String name;
  private int balance;

  public Member (String name, int balance) {

  public String getName () …
  public void setName () …

  public boolean contribute (int amount) …
  public int getBalance () …
```

Cohesion is Good

Cohesion also applies at the level of methods.

A method should do exactly one, well defined, thing.

Example:

- A "deposit" method should handle the logic of a deposit.
- A "deposit" method should not print messages, request new paying-in books, recommend new accounts …

Coupling

Coupling

The concept of *coupling* relates to how closely connected two classes are.

Coupling

The concept of *coupling* relates to how closely connected two classes are.

Tightly Coupled classes are closely related, probably by calling each other's methods.

In this case, a change to one class could well imply the need for a change in the other.

Coupling

The concept of *coupling* relates to how closely connected two classes are.

Tightly Coupled classes are closely related, probably by calling each other's methods.

In this case, a change to one class co
change in the other.

It follows that tight coupling is to be avoided if at all possible.

Coupling

The concept of *coupling* relates to how closely connected two classes are.

Tightly Coupled classes are closely related, probably by calling each other's methods.

In this case, a change to one class co
change in the other.

Tightly coupled classes are unlikely to provide many opportunities for reuse.

Coupling

The concept of *coupling* relates to how closely connected two classes are.

Tightly Coupled classes are closely related, probably by calling each other's methods.

In this case, a change to one class co
change in the other.

In the worst case, consider what happens if A is tightly coupled with B, which is tightly coupled with C, which is tightly coupled …

Coupling

The concept of *coupling* relates to how closely connected two classes are.

Loosely Coupled classes may well interact, but are not reliant on the details of each other.

Ideally, changes in one class would not require changes in another.

Coupling

The concept of *coupling* relates to how closely connected two classes are.

An analogy:

- The components of a Hi-Fi are loosely coupled, meaning that one can be replaced without affecting the others.
- Components in an all-in-one music system are tightly coupled, so if one fails the whole system is borked.

Good Design

Generally, we say that good OO design exhibits:

- High cohesion in the individual classes.
- Low coupling between classes.

Good Design

Generally, we say that good OO design exhibits:

- High cohesion in the individual classes.
- Low coupling between classes.

And the same applies to methods within a class.

Good Design

Generally, we say that good OO design exhibits:

- High cohesion in the individual classes.
- Low coupling between classes.

And the same applies to methods within a class.

This is kind of why you do "Design" alongside "Development" in this module, of course.

Interacting Classes

Interacting Classes

But obviously we won't get anywhere if classes cannot interact at all.

Low Coupling - High Cohesion

So what classes can be used in another class?

Interacting Classes

So what classes can be used in another class?

A class can use:

- Any class defined in the same package.
- Any class defined in the Java libraries (with an import).
- Any class defined in a related package (with an import).
- Any class defined anywhere (with a download and an import).

Interacting Classes

So what classes can be used in another class?

A class can use:

- Any class defined in the same package.
- Any class defined in the Java
- Any class defined in a related
- Any class defined anywhere (
  import).

It neatly follows that related classes belong in the same package.

Interacting Classes

A `Bank` class can use a `BankAccount` class defined in the same package.

`Bank.java`
`BankAccount.java`

Interacting Classes

A Bank class can use a BankAccount class defined in the same package.

The Bank class can create BankAccount objects.

Bank.java

```
BankAccount ba = new BankAccount ();
```

BankAccount.java

Interacting Classes

A Bank class can use a BankAccount class defined in the same package.

The Bank class can create BankAccount objects.

Note that BankAccount is completely unaware of Bank.

BankAccount is therefore potentially reusable.

Bank.java

```
BankAccount ba = new BankAccount ();
```

BankAccount.java

Interacting Classes

A Bank class can use a BankAccount class defined in the same package.

The Bank class can create BankAccount objects.

Bank may also use ArrayLists, provided the class is imported.

(ArrayList has no idea that it is going to be used with BankAccount objects.)

Bank.java

```
import java.util.ArrayList;

private accounts ArrayList <BankAccount>;

BankAccount ba = new BankAccount ();
```

BankAccount.java

Non-standard Classes

There are many, many, other classes out there.

See for example Apache's `StringUtils`:

https://commons.apache.org/proper/commons-lang/apidocs/org/apache/commons/lang3/StringUtils.html

Libraries such as this are often referred to as *dependencies*, which can be managed by special tools (Maven, Gradle).

toString

toString

The special `toString` method returns a String representation of an object.

It can be in any format the programmer requires.

```
public class Employee {

  private String name;
  private int rating;

.
.
.

  public String toString () {

    String s = "";

    s += this.name + " is ";
    s += this.getRatingDescription ();

    return s;
  }
```

toString

The special `toString` method returns a String representation of an object.

It can be in any format the programmer requires.

(If you were looking closely earlier on you will have seen that Python has one too, called `__str__`).

```
public class Employee {

  private String name;
  private int rating;

.
.
.

  public String toString () {

    String s = "";

    s += this.name + " is ";
    s += this.getRatingDescription ();

    return s;
  }
```

Static Methods

Static Context

There will come a time when your IDE will tell you that you cannot use a "non-static method" in a "static context".

Static Context

There will come a time when your IDE will tell you that you cannot use a "non-static method" in a "static context".

The fix is **not** to insert `static` in the method signature, unless this is what you need, and you understand what it does.

Really.

I mean that.

Static Methods

All the methods we have written so far (except one) were invoked on objects.

A `static` method is not invoked on an object.

Think of it as just a handy bit of code you want to give a name to.

Hello World

```
package week10.greeters;

public class ObjectGreeter {

     public ObjectGreeter () {}

     public void sayHelloWorld () {
         System.out.println ("Hello, World");
     }

     public static void main (String[] args) {

         ObjectGreeter og = new ObjectGreeter ();
         og.sayHelloWorld ();

     }
}
```

Static Hello World

```
package week10.greeters;

public class StaticGreeter {

     public static void sayHelloWorld () {
         System.out.println ("Hello, World");
     }

     public static void main (String[] args) {
         sayHelloWorld ();
     }
}
```

Library Classes: The Scanner

Keyboard Input

The `Scanner` class provides mechanisms to read input from the keyboard.

Keyboard Input

The `Scanner` class provides mechanisms to read input from the keyboard.

Actually, it reads from *streams*. Streams might be the keyboard, a file, or anything else that generates data.

Keyboard Input

The `Scanner` class provides mechanisms to read input from the keyboard.

Actually, it reads from *streams*. Streams might be the keyboard, a file, or anything else that generates d

`System.out` we have met.
`System.in` is the keyboard (usually).
`System.err` is the error stream.

Static Hello World Revisited

```
package week10.greeters;

import java.util.Scanner;

public class NamedGreeter {

     public static void sayHelloWorld (String name) {
         System.out.println ("Hello, " + name);
     }

     public static void main (String[] args) {

         Scanner in = new Scanner (System.in);

         System.out.print ("Hello. Who are you? ");
         String name = in.nextLine ();

         sayHelloWorld (name);
     }
}
```

// Comments

Comments

The Elder Rule of Comments

Include a comment if some code needs explanation.

Then look at the code really, really, carefully to see if you can refactor it to remove the need for a comment.

If the comment is still there, look again.

Comments

The Elder Rule of Comments

Include a comment if some code needs explanation.

Then look at the code really, really, carefully to see if you can refactor it to remove the need for a comment.

If the comment is still there, look again.

```
/*
 * This method always returns true.
 */

public boolean santaExists () {
  return false;
}
```

Comments

The Elder Rule of Comments

Include a comment if some code needs explanation.

Then look at the code really, really, carefully to see if you can refactor it to remove the need for a comment.

If the comment is still there, look again.

```
/*
 * This method always returns true.
 */

public boolean santaExists () {
  return false;
}
```

IntelliJ Demo Time