University of HUDDERSFIELD

CFS2160: Software Design and Development

Week 14: Patterns

And generally being neat.

Steve McGuire
s.mcguire@hud.ac.uk

An Example

Poppleton Dogs Home is raising funds.

A system is required that will store details of all the fundraisers, and the amount they have raised.

The system should also produce a neat report of all the amounts raised, along with the total amount.

The report should be sorted with the top fundraisers listed first.

An Example

Poppleton Dogs Home is raising funds.

A system is required that will store details of all the fundraisers, and the amount they have raised.

The system should also produce a ne
raised, along with the total amount.

The report should be sorted with the

Have we seen this problem before?

Abstraction

This problem is actually Cracker Packer in disguise.

We can therefore *abstract* from the solution to that problem to start to build a new solution.

Abstraction

This problem is actually Cracker Packer in disguise.

We can therefore *abstract* from the solution to that problem to start to build a new solution.

We can probably also expect to be able to reuse some of the code from that project, probably with a few cosmetic changes.

Patterns

Abstracting further, we can generalise.

This problem involves a basic class, another class that contains a collection of these and a class for the Main method where the instances of classes are created.

It's a classic *Design Pattern*.

Design

We will therefore have a `FundRaiser` class.

A `FundRaisingTeam` class will contain a collection of these.

And a class will be needed which has the Main method and is used to create the objects and generate the required report - call it `Campaign`.

Design

We will therefore have a `FundRaiser` class.

A `FundRaisingTeam` class will contain a collection of these.

And a class will be needed which has the Main method and is used to create the objects and gener
it `Campaign`.

Reminder

We design for high cohesion.
(Each class should do just its job well.)

Design

We will therefore have a `FundRaiser` class.

A `FundRaisingTeam` class will contain a collection of these.

And a class will be needed which has the Main method and is used to create the objects and gener
it `Campaign`.

Reminder

We design for low coupling.
(Classes do not need to know of or rely on each other)

Design

We will therefore have a `FundRaiser` class.

A `FundRaisingTeam` class will contain a collection of these.

And a class will be needed which has the Main method and is used to create the objects and gener
it `Campaign`.

Reminder

We believe in encapsulation.
(Meaningful getters and setters.
Correct access modifiers)

An Example

Poppleton Dogs Home is raising funds.

A system is required that will store details of all the fundraisers, and the amount they have raised.

The system should also produce a **neat** report of all the **amounts raised**, along with the total amount.

The report should be **sorted** with the top fundraisers listed first.

An Example

Poppleton Dogs Home is raising funds.

A system is required that will store details of all the fundraisers, and the amount they have raised.

The system should also produce a **neat** report of all the **amounts raised**, along with the total amount.

The report should be **sorted** with the *top fundraisers listed first*.

FundRaiser

This will need:

- Instance variables for name, amount raised and probably an ID.
- Getters and setters for the above.
- A method to print the details out neatly formatted.
- Maybe some way to allow fundraisers to be sorted?

FundRaisingTeam

This will need:

- Instance variable for a collection of `FundRaiser` objects.
  - An `ArrayList` will be fine.
- Methods to add and maybe delete team members.
- A method to print out the whole team, neatly.
- A method to find the total raised.
- Some way to sort the team in order of amount raised.

Campaign

This will need:

- A Main method as the starting point.
- To create objects to represent all the fundraisers.
- To add them to the team.
- To print the report as required.

Issues

So, the new issues that we need to be able to program are:

- How to print an amount of money?
- How to print a neat table?
- How to sort an `ArrayList` of objects?

Issues

So, the new issues that we need to be able to program are:

- How to print an amount of money?
- How to print a neat table?
- How to sort an `ArrayList`

The message is going to be the same with all three of these, but we'll start with the simplest.

Printing Money

We need to store "money" in some way that means we can do arithmetic on it.

```
private double cash;
```

Printing Money

We need to store "money" in some way that means we can do arithmetic on it.

But when we print it out, we want a currency symbol, which makes the output a String.

```
private double cash;
```

Printing Money

We need to store "money" in some way that means we can do arithmetic on it.

But when we print it out, we want a currency symbol, which makes the output a String.

We could try something like this.

```
private double cash;

System.out.print ("£" + cash);
```

Printing Money

We need to store "money" in some way that means we can do arithmetic on it.

But when we print it out, we want a currency symbol, which makes the output a String.

We could try something like this.

```
private double cash;

System.out.print ("£" + cash);
```

This could well work.
But, imagine changing the program to work with a different currency.

Printing Money

We need to store "money" in some way that means we can do arithmetic on it.

But when we print it out, we want a currency symbol, which makes the output a String.

We could try something like this.

```
private double cash;

System.out.print ("£" + cash);
```

This could well work.
And what would happen if the value in `cash` had four decimal places, or none at all?

Printing Money

We need to store "money" in some way that means we can do arithmetic on it.

So we don't do this.

We investigate the Java libraries, because there is bound to be something there that will do all this for us.

```
private double cash;
```

~~`System.out.print ("£" + cash);`~~

Printing Money

We need to store "money" in some way that means we can do arithmetic on it.

So we don't do this.

We investigate the Java libraries, because there is bound to be something there that will do all this for us.

```
private double cash;
```

~~`System.out.print ("£" + cash);`~~

This is the message!
One of the tricks of programming well is to use stuff that is already there, and is tried and tested.

Printing Money

We need to store "money" in some way that means we can do arithmetic on it.

Googling would lead us to a much neater solution.

We create a `NumberFormat` object, and pass it another object that says we want the format for UK Currency.

```
private double cash;
```

```
NumberFormat money =
NumberFormat.getCurrencyInstance (Locale.UK);
```

Printing Money

We need to store "money" in some way that means we can do arithmetic on it.

Googling would lead us to a much neater solution.

We create a `NumberFormat` object, and pass it another object that says we want the format for UK Currency.

And we use this to print the money in the required format.

```
private double cash;

NumberFormat money =
NumberFormat.getCurrencyInstance (Locale.UK);


cash = 12.99;
System.out.println (money.format (cash));

// Prints "£12.99".
```

Printing Money

We need to store "money" in some way that means we can do arithmetic on it.

This will always maintain the required formats for the *locale*.

```
private double cash;
```

```
NumberFormat money =
NumberFormat.getCurrencyInstance (Locale.UK);
```

```
cash = 12.99;
System.out.println (money.format (cash));

// Prints "£12.99".
```

Printing Money

We need to store "money" in some way that means we can do arithmetic on it.

This will always maintain the required formats for the *locale*.

So our code could easily be tweaked for use in, say, France.

```
private double cash;

NumberFormat money =
NumberFormat.getCurrencyInstance
                                (Locale.FRANCE);

cash = 12.99;
System.out.println (money.format (cash));

// Prints "12,99 €".
```

Printing Money

We need to store "money" in some way that means we can do arithmetic on it.

This will always maintain the required formats for the *locale*.

So our code could easily be tweaked for use in, say, France.

```
private double cash;

NumberFormat money =
NumberFormat.getCurrencyInstance
                                (Locale.FRANCE);

cash = 12.99;
System.out.println (money.format (cash));
```

All this requires at least two `import` statements in the class. IntelliJ is really good at keeping track of this for you.

Printing Money

We need to store "money" in some way that means we can do arithmetic on it.

This will always maintain the required formats for the *locale*.

So our code could easily be tweaked for use in, say, France.

```
private double cash;

NumberFormat money =
NumberFormat.getCurrencyInstance
                                (Locale.FRANCE);

cash = 12.99;
System.out.println (money.format (cash));
```

Our Googling would also have uncovered the `DecimalFormat` class, but this is a less good solution because it doesn't handle the currency symbol.

Printing a Table

How to print a neat table?

We will need to specify:

- Width of columns.
- Justified left, right, or centred.
- (Possibly) formats for numbers.

Printing a Table

How to print a neat table?

We will need to specify:

- Width of columns.
- Justified left, right, or centred.
- (Possibly) formats for numbers.

As with many things Java, there are several ways to do this.
We'll pick a simple one that works for tables.
See `StringBuilder` for an alternative.

Printing a Table

How to print a neat table?

The `format` method of the `String` class takes a "format string" and formats the other arguments accordingly to make a new String.

Printing a Table

How to print a neat table?

The `format` method of the `String` class takes a "format string" and formats the other arguments accordingly to make a new String.

The two %s are substituted for the strings ‘x’ and ‘y’.

%s expects a String to substitute.

Other types can be used

```
String s;

s = String.format ("%s %s", "x", "y");

// s is "x y".
```

Printing a Table

How to print a neat table?

The `format` method of the `String` class takes a "format string" and formats the other arguments accordingly to make a new String.

The format string can contain field width.

```
String s;

s = String.format ("%10s", "x");

// s is "         x".
```

Printing a Table

How to print a neat table?

The `format` method of the `String` class takes a "format string" and formats the other arguments accordingly to make a new String.

The format string can contain field width.

And justification.

```
String s;

s = String.format ("%-10s", "x");

// s is "x          ".
```

Printing a Table

How to print a neat table?

The `format` method of the `String` class takes a "format string" and formats the other arguments accordingly.

The format string can contain field width.

And justification.

```
String s;

s = String.format ("%-10s", "x");

// s is "x         ".
```

This is actually all we need for this task since all the output is strings. Check the docs for more details.

Sorting

So, how do we sort things?

Unsurprisingly, if our collection contained numeric data, sorting would "just work".

But in this program our collection will contain objects, representing fundraisers …

Sorting

So, how do we sort things?

Unsurprisingly, if our collection contained numeric data, sorting would "just work".

But in this program our collection will contain objects, representing fundraisers …

Warning
We are about to touch on a very new (to us) bit of Java: interfaces. We will gloss over details, and return to it later on.

Sorting

So, how do we sort things?

Unsurprisingly, if our collection contained numeric data, sorting would "just work".

But in this program our collection will contain objects, representing fundraisers …

```
FundRaiser f1 = new FundRaiser ("Fred", 10);
FundRaiser f2 = new Fundraiser ("June", 20);
```

Sorting

So, how do we sort things?

Unsurprisingly, if our collection contained numeric data, sorting would "just work".

But in this program our collection will contain objects, representing fundraisers …

Which are in a collection …

```
FundRaiser f1 = new FundRaiser ("Fred", 10);
FundRaiser f2 = new Fundraiser ("June", 20);

theTeam.addFundRaiser (f1);
theTeam.addFundRaiser (f2);
```

Sorting

So, how do we sort things?

Unsurprisingly, if our collection contained numeric data, sorting would "just work".

But in this program our collection will contain objects, representing fundraisers …

Which are in a collection …

```
FundRaiser f1 = new FundRaiser ("Fred", 10);
FundRaiser f2 = new Fundraiser ("June", 20);

theTeam.addFundRaiser (f1);
theTeam.addFundRaiser (f2);
```

It is obvious that June should be "sorted" before Fred.
So what piece of information does Java need in order to do that for us?

Sorting

So, how do we sort things?

Unsurprisingly, if our collection contained numeric data, sorting would "just work".

But in this program our collection will contain objects, representing fundraisers …

Which are in a collection …

```
FundRaiser f1 = new FundRaiser ("Fred", 10);
FundRaiser f2 = new Fundraiser ("June", 20);

theTeam.addFundRaiser (f1);
theTeam.addFundRaiser (f2);
```

We need to define how to *compare* two `FundRaiser` objects. That is, how to tell if they are "equal", "less than" or "greater than" one another.

Sorting

So, how do we sort things?

Unsurprisingly, if our collection contained numeric data, sorting would "just work".

But in this program our collection will contain objects, representing fundraisers …

Which are in a collection …

```
FundRaiser f1 = new FundRaiser ("Fred", 10);
FundRaiser f2 = new Fundraiser ("June", 20);

theTeam.addFundRaiser (f1);
theTeam.addFundRaiser (f2);
```

We need to define how to *compare* two `FundRaiser` objects. Pedantically, any two of "less than", "equal to" or "greater than" will do.

Comparable

To do this, we declare that the `FundRaiser` class implements `Comparable` behaviour.

```
public class FundRaiser implements
             Comparable <FundRaiser> {
```

Comparable

To do this, we declare that the `FundRaiser` class implements `Comparable` behaviour.

```
public class FundRaiser implements
             Comparable <FundRaiser> {
```

`Comparable` is an *interface* which defines the methods that must exist in order for a class to be "sortable". Without the required methods the programme wont work.

Comparable

To do this, we declare that the `FundRaiser` class implements `Comparable` behaviour.

We then add a method called `compareTo` which does the comparison.

The return value indicates how the two objects should be ordered.

```
public class FundRaiser implements
             Comparable <FundRaiser> {

@Override
public int compareTo (FundRaiser fr) {
     if (fr.amount > this.amount) {
          return 1;
     }
     else if (fr.amount < this.amount) {
          return -1;
     }
     else {
          return 0;
     }
}
```

Comparable

To do this, we declare that the `FundRaiser` class implements `Comparable` behaviour.

We then add a method called

Remember that here we are sorting the higher values to the top so the final list in the report is in *descending* order.
It's a bit mind-bending.

```
public class FundRaiser implements
             Comparable <FundRaiser> {

@Override
public int compareTo (FundRaiser fr) {
     if (fr.amount > this.amount) {
          return 1;
     }
     else if (fr.amount < this.amount) {
          return -1;
     }
     else {
          return 0;
     }
}
```

Comparable

To do this, we declare that the `FundRaiser` class implements `Comparable` behaviour.

We then add a method called

Again, we don't need to know how Java does the sorting, we just need to know how to implement the Comparable interface and its methods

```
public class FundRaiser implements
             Comparable <FundRaiser> {

@Override
public int compareTo (FundRaiser fr) {
     if (fr.amount > this.amount) {
          return 1;
     }
     else if (fr.amount < this.amount) {
          return -1;
     }
     else {
          return 0;
     }
}
```

Sorting

So, how do we sort things?

Once we have defined how objects in the class are ordered (or compared), we call the `Collections.sort()` method and sorting will "just work".

```
FundRaiser f1 = new FundRaiser ("Fred", 10);
FundRaiser f2 = new Fundraiser ("June", 20);

theTeam.addFundRaiser (f1);
theTeam.addFundRaiser (f2);

Collections.sort (theTeam);
```

Sorting

So, how do we sort things?

Once we have defined how objects in the class are ordered (or compared), sorting will "just work".

`Collections.sort` is a static method from the `Collections` class.

It'll sort pretty much anything as long as it knows how to compare.

```
FundRaiser f1 = new FundRaiser ("Fred", 10);
FundRaiser f2 = new Fundraiser ("June", 20);

theTeam.addFundRaiser (f1);
theTeam.addFundRaiser (f2);

Collections.sort (theTeam);
```

Sorting

So, how do we sort things?

Once we have defined how objects in the class are ordered (or compared), sorting will "just work".

Again, suitable `import` statements are needed:

`java.util.Collections`

```
FundRaiser f1 = new FundRaiser ("Fred", 10);
FundRaiser f2 = new Fundraiser ("June", 20);

theTeam.addFundRaiser (f1);
theTeam.addFundRaiser (f2);

Collections.sort (theTeam);
```

FundRaiser

```
public class FundRaiser implements Comparable <FundRaiser> {

    private String name;
    private String id;
    private double amountRaised;

.
.
.

}
```

FundRaiser

```
public class FundRaiser implements Comparable <FundRaiser> {

    private String name;
    private String id;
    private double amountRaised;

    public void printFormatted () {

        NumberFormat gb = NumberFormat.getCurrencyInstance (Locale.UK);

        final String formatString = "%-4s %-12s %8s";

        System.out.println (String.format (formatString, this.id, this.name,
                                           gb.format (this.amountRaised)));
     }

}
```

FundRaisingTeam

```
public class FundRaisingTeam {

    private static final String TEAM_NAME = "Poppleton Dogs Home";

    private ArrayList <FundRaiser> fundRaisers;

.
.
.
}
```

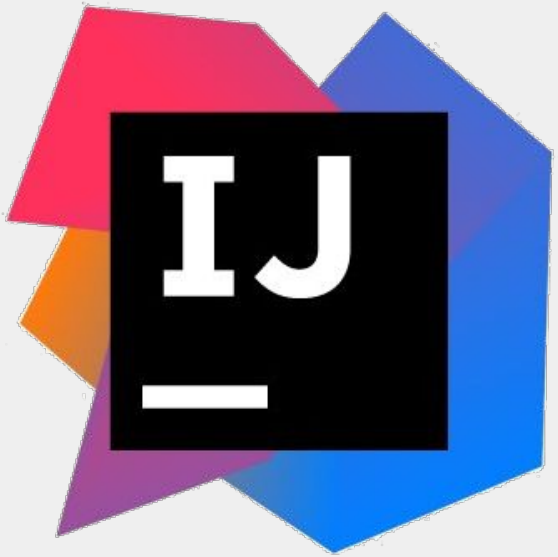
FundRaisingTeam

```
public class FundRaisingTeam {

    private static final String TEAM_NAME = "Poppleton Dogs Home";

    private ArrayList <FundRaiser> fundRaisers;

    public void sortTeam () {
        Collections.sort (this.fundRaisers);
    }

}
```

FundRaisingTeam

```
public class FundRaisingTeam {

    private static final String TEAM_NAME = "Poppleton Dogs Home";

    private ArrayList <FundRaiser> fundRaisers;

    public void printTeam () {

        if (this.fundRaisers.isEmpty ()) {
            System.out.println ("No fundraisers in this team.");
        }
        else {
            for (FundRaiser fr : fundRaisers) {
               fr.printFormatted ();
        }
    }
}

}
```

IntelliJ Demo Time



Git

```
https://github.com/TonyJenkins/fundraiser-demo.git
```