University of HUDDERSFIELD

CFS2160: Software Design and Development

Lecture 8: Class Definitions

Classes. Attributes. Methods.

Tony Jenkins
A.Jenkins@hud.ac.uk

Status Report

We have:

- Written at least two Java programs.
- Mastered how to run them, and find the output.

More specifically, we have:

- Used the `main` method inside a class to use the other methods of the class to manipulate the fields to form a simple program.

Basic Class Structure

```
public class Student
{
    // Inner part of class omitted.
}
```

Basic Class Structure

```
public class Student
{
    // Inner part of class omitted.
}
```

Basic Class Structure

```
public class Student
{
    // Fields (attributes, instance variables)
    // Constructor(s)
    // Methods
}
```

Basic Class Structure

```
public class Student
{
    // Fields (attributes, instance variables)
    // Constructor(s)
    // Methods
}
```

The values of the instance variables define the object's current *state*.

Basic Class Structure

```
public class Student
{
    // Fields (attributes, instance variables)
    // Constructor(s)
    // Methods
}
```

All objects in the class have the same instance variables.
But, probably, they have different state.

Basic Class Structure

```
public class Student
{
    // Fields (attributes, instance variables)
    // Constructor(s)
    // Methods
}
```

Methods implement the object's behaviours.
They may, or may not, return a value.

Basic Class Structure

```
public class Student
{
    // Fields (attributes, instance variables)
    // Constructor(s)
    // Methods
}
```

Methods implement the object's behaviours.
Methods that do not return a value are defined to return `void`.

Basic Class Structure

```
public class Student
{
    // Fields (attributes, instance variables)
    // Constructor(s)
    // Methods
}
```

The Constructor is a special method that creates an object.

Fields

```
private boolean hasRegistered;
```

- Visibility Modifier.
  - Fields are usually `private`.
- Data Type.
  - This field is a Boolean "true/false" value.
- Variable Name.
  - Meaningful, and starting with `lowerCase` letter.

Methods

```
public String getName ()
```

- Visibility Modifier.
  - Methods are usually `public`.
- Data Type.
  - This method returns a `String`.
- Method Name.
  - Meaningful, and starting with `lowerCase` letter.

Methods

```
public void setName (String newName)
```

- Visibility Modifier.
  - Methods are usually `public`.
- Data Type.
  - This method returns nothing.
- Method Name.
  - Meaningful, and starting with `lowerCase` letter.

Methods

```
public void setName (String newName)
```

- Visibility Modifier.
  - Methods are usually `public`.
- Data Type.
  - This method returns nothing.
- Method Name.
  - Meaningful, and starting with `lowe`

This method also has a single parameter.
It is a `String`, called `newName`.

Methods

```
public void setName (String newName)
```

This is the *signature* of the method.

It specifies everything needed to use the method: so it is what you will find in the documentation.

Every method in a class must have a unique signature.

The Docs

Java is big.

The Java API docs are here:

https://docs.oracle.com/javase/8/docs/api/

They are big.

Very big.

The Docs

Java is big.

The Java API docs are here:

https://docs.oracle.com/javase/8/docs/api/

They are big.

Very big.

A key feature of Python is that there should be one - and only one - way to do something. Java is not like that.

Constructor

Every class must have a "constructor".

This special method is responsible for creating an instance of the class (an object, if you will).

A constructor may be provided with initial values for the fields.

Or it may set some defaults.

Or it can even do a mixture of the two.

Constructor

The constructor has the same name as the class.

It must also be `public`.

```
public Student (String name, int age)
{
    this.name = name;
    this.age = age;
    hasRegistered = false;
}
```

Constructor

The constructor has the same name as the class.

It must also be `public`.

```
public Student (String name, int age)
{
    this.name = name;
    this.age = age;
    hasRegistered = false
}
```

Why `public`?
This means it can be used by other programs and classes.

Constructor

The constructor has the same name as the class.

It must also be `public`.

```
public Student (String name, int age)
{
    this.name = name;
    this.age = age;
    hasRegistered = false
}
```

Why `public`?
`private` methods can only be used in the class where defined.

Constructor

The constructor has the same name as the class.

It must also be `public`.

```
public Student (String name, int age)
{
    this.name = name;
    this.age = age;
    hasRegistered = false;
}
```

The `this` keyword is used to remove ambiguity if the parameter and the instance variable have the same name.

Constructors

There can be several constructors in a class, as long as they have different *signatures*.

```
public Student (String name, int age)

public Student (String name, int age, bool registered)

public Student ()
```

Methods

Likewise, there can be several methods with the same name, as long as they have different *signatures*.

```
public void addInterest (int interestRate)

public double addInterest (int interestRate, boolean extra)

public boolean addInterest (int interestRate, int limit)
```

The Bank Account

Last week's `BankAccount` class had four instance variables.

```
private String accountNumber;
private String accountHolder;
private double balance;
private boolean hasOverdraft;
```

The Bank Account

Last week's `BankAccount` class had four instance variables.

`double` is a floating-point number.

```
private String accountNumber;
private String accountHolder;
private double balance;
private boolean hasOverdraft;
```

The Bank Account

Last week's `BankAccount` class had four instance variables.

`double` is a floating-point number.

Java provides a bunch of *primitive types*:

`int`, `double`, `boolean` are good.

`String` is slightly different (spot the capital S).

```
private String accountNumber;
private String accountHolder;
private double balance;
private boolean hasOverdraft;
```

The Bank Account

Last week's `BankAccount` class had four instance variables.

Each instance variable had a getter and a setter ("accessor" and "mutator").

```
public double getBalance () {
    return balance;
}

public void setBalance (double balance)
{
    this.balance = balance;
}
```

The Bank Account

Last week's `BankAccount` class had four instance variables.

Each instance variable had a getter and a setter ("accessor" and "mutator").

Remember that IntelliJ will write these for you.

```
public double getBalance () {
    return balance;
}

public void setBalance (double balance)
{
    this.balance = balance;
}
```

The Bank Account

Last week's `BankAccount` class had four instance variables.

Each instance variable had a getter and a setter ("accessor" and "mutator").

Remember that IntelliJ will write these for you.

(In the "real world" we use a tool that takes even this step out!)

```
public double getBalance () {
    return balance;
}

public void setBalance (double balance)
{
    this.balance = balance;
}
```

The Bank Account

Last week's `BankAccount` class had four instance variables.

Each instance variable had a getter and a setter ("accessor" and "mutator").

By custom, these are called "get" and "set", followed by the name of the field.

Alternatively, "has" can work better for Boolean values.

```
public double getBalance () {
    return balance;
}

public void setBalance (double balance)
{
    this.balance = balance;
}
```

The Bank Account

Last week's `BankAccount` class had four instance variables.

Each instance variable had a getter and a setter ("accessor" and "mutator").

By custom, these are called "get" and "set", followed by the name of the field.

Alternatively, "has" can work better for Boolean values.

```
public double getBalance () {
    return balance;
}

public void setBalance (double balance)
{
```

The names matter.
We are aiming for code that speaks for itself.
Comments are bad.

The Bank Account

There were also methods to implement the various behaviours of a bank account.

Like withdrawing, and depositing.

```
public void deposit (double amount) {
    this.balance += amount;
}

public void withdraw (double amount) {
     this.balance -= amount;
}
```

The Bank Account

There were *two* constructors.

One took an initial value for the balance, the other just set it to zero.

Which was which is obvious from the different signatures.

```
public BankAccount
       (String accountNumber,
        String accountHolder,
        boolean hasOverdraft) {

public BankAccount
       (String accountNumber,
        String accountHolder,
        double balance,
        boolean hasOverdraft) {
```

The Bank Account

There were *two* constructors.

One took an initial value for the balance, the other just set it to zero.

Which was which is obvious from the different signatures.

And which was required was obvious from the way in which the constructor was called.

```
public BankAccount
       (String accountNumber,
        String accountHolder,
        boolean hasOverdraft) {

public BankAccount
       (String accountNumber,
        String accountHolder,
        double balance,
        boolean hasOverdraft) {
```

The Bank Account

There was also a special method called `toString`.

This provides (returns) a string representation of the object, and is called by magic when the object is printed.

```
BankAccount b = new BankAccount
                ("78325123",
                 "Richard Branson",
                 false);

System.out.println (b);
```

The Bank Account

There was also a special method called `toString`.

This provides (returns) a string representation of the object, and is called by magic when the object is printed.

By magic!

```
BankAccount b = new BankAccount
                ("78325123",
                 "Richard Branson",
                 false);

System.out.println (b);
System.out.println (b.toString ());
```

The Bank Account

To invoke any other methods, we use the object name, followed by the method name …

… with any parameters in brackets.

```
BankAccount b = new BankAccount
                ("78325123",
                 "Richard Branson",
                 false);

System.out.println (b);

b.setBalance (1000.0);
b.deposit (150.0);
b.addInterest (5);
```

The Bank Account

There were also methods to implement the various behaviours of a bank account.

Like withdrawing, and depositing.

These two methods are actually not very satisfactory.

```
public void deposit (double amount) {
    this.balance += amount;
}

public void withdraw (double amount) {
     this.balance -= amount;
}
```

The Bank Account

There were also methods to implement the various behaviours of a bank account.

Like withdrawing, and depositing.

These two methods are actually not very satisfactory.

What would happen?:

```
d.deposit (-150);
```

```
public void deposit (double amount) {
    this.balance += amount;
}

public void withdraw (double amount) {
     this.balance -= amount;
}
```

The Bank Account

There were also methods to implement the various behaviours of a bank account.

Like withdrawing, and depositing.

These two methods are actually not very satisfactory.

What would happen?:

```
d.deposit (-150);
```

```
public void deposit (double amount) {
    this.balance += amount;
}

public void withdraw (double amount) {
```

So, we need to check the value in the parameter before allowing the method to proceed.

The Bank Account

There were also methods to implement the various behaviours of a bank account.

Like withdrawing, and depositing.

These two methods are actually not very satisfactory.

What would happen?:

```
d.deposit (-150);
```

```
public void deposit (double amount) {
    this.balance += amount;
}

public void withdraw (double amount) {
```

So, we need to check the value in the parameter before allowing the method to proceed.
Note that in Java we *do not* need to check what type it is.

Conditional Statements

We need to implement this rule in the `deposit` method:

"If the value provided is greater than zero, add it to the balance. Otherwise do nothing at all."

This is a simple *conditional* statement.

Conditional Statements

```
if (some test condition is true)
{
    // Do these statements.
}
else
{
    // Do these statements.
}
```

A Note on Layout

Indentation has no influence on program structure in Java.

But code is still indented to show the structure (IntelliJ will do this for you).

A conditional can be written like so:

```
if (amount > 0)
{
```

A Note on Layout

Indentation has no influence on program structure in Java.

But code is still indented to show the structure (IntelliJ will do this for you).

Or, equally, like so:

```
if (amount > 0) {
```

A Note on Layout

Indentation has no influence on program structure in Java.

But code is still indented to show the structure (IntelliJ will do this for you).

Or, equally, like so:

```
if (amount > 0) {
```

It doesn't matter which you do, as long as you're consistent.
(I will generally do this one.)

A Note on Layout

Indentation has no influence on program structure in Java.

But code is still indented to show the structure (IntelliJ will do this for you).

Or, equally, like so:

```
if (amount > 0) {
```

It doesn't matter which you do, as long as you're consistent.
(I will generally do this one.)
It makes the program shorter.

Protecting Fields

By validating the deposit amount, we are protecting the integrity of the value in the balance field.

Protecting Fields

By validating the deposit amount, we are protecting the integrity of the value in the balance field.

This version does not do this, because negative deposits would be allowed.

```
public void deposit (double amount) {
    this.balance += amount;
}
```

Protecting Fields

By validating the deposit amount, we are protecting the integrity of the value in the balance field.

This version does not do this, because negative deposits would be allowed.

Deposits of zero are also allowed, which are meaningless, but at least would do no harm!

```
public void deposit (double amount) {
    this.balance += amount;
}
```

Protecting Fields

By validating the deposit amount, we are protecting the integrity of the value in the balance field.

This version is better. Attempted deposits of zero (or less) will have no effect on the balance.

```
public void deposit (double amount) {

    if (amount > 0.0) {
        this.balance += amount;
    }

}
```

Protecting Fields

By validating the deposit amount, we are protecting the integrity of the value in the balance field.

This version is better. Attempted deposits of zero (or less) will have no effect on the balance.

The remaining issue here is now that whatever called the method has no idea that its call failed.

```
public void deposit (double amount) {

    if (amount > 0.0) {
        this.balance += amount;
    }

}
```

Protecting Fields

By validating the deposit amount, we are protecting the integrity of the value in the balance field.

A naive approach would be to hopefully print a message to say that the deposit failed.

```
public void deposit (double amount) {

    if (amount > 0.0) {
        this.balance += amount;
    }
    else {
        System.out.println ("Failed");
    }

}
```

Protecting Fields

By validating the deposit amount, we are protecting the integrity of the value in the balance field.

This is bad.

- The method now does two distinct things.
- It is only ever useful if there is a human reading the result from every bank deposit.

```
public void deposit (double amount) {

    if (amount > 0.0) {
        this.balance += amount;
    }
    else {
        System.out.println ("Failed");
    }

}
```

Protecting Fields

By validating the deposit amount, we are protecting the integrity of the value in the balance field.

Much better is to provide an indication to whatever called the method that all was not well (or that it was well).

```
public void deposit (double amount) {

    if (amount > 0.0) {
        this.balance += amount;
    }
    else {
        System.out.println ("Failed");
    }

}
```

Protecting Fields

By validating the deposit amount, we are protecting the integrity of the value in the balance field.

Much better is to provide an indication to whatever called the method that all was not well (or that it was well).

So we change the method's return type, and send back an indication of success or failure.

```
public boolean deposit (double amount) {

    if (amount > 0.0) {
        this.balance += amount;
        return true;
    }
    else {
        return false;
    }

}
```

Checking for Success

So now when using the `deposit` method, a program can check if the deposit worked.

```
BankAccount b = new BankAccount (...);
```

Checking for Success

So now when using the `deposit` method, a program can check if the deposit worked.

Or, if this does not matter, the return value can be ignored.

So this still works.

```
BankAccount b = new BankAccount (...);

b.deposit (150.0);
```

Checking for Success

So now when using the deposit method, a program can check if the deposit worked.

The call to the method can be used as the value to be tested in a conditional statement, like this.

```
BankAccount b = new BankAccount (...);

if (b.deposit (150.0)) {
    // It worked.
}
else {
    // It didn't work.
}
```

Checking for Success

So now when using the deposit method, a program can check if the deposit worked.

The call to the method can be used as the value to be tested in a conditional statement.

Most usually, it is only the failure case that matters.

```
BankAccount b = new BankAccount (...);

if (!b.deposit (150.0)) {
    // It didn't work.
}


// Above is the same as:
//   if (b.deposit (150.0) == false) {
// Only shorter.
```

Checking for Success

So now when using the deposit method, a program can check if the deposit worked.

The call to the method can be used as the value to be tested in a conditional statement.

Most usually, it is only the failure case that matters.

```
BankAccount b = new BankAccount (...);

if (!b.deposit (150.0)) {
    // It didn't work.
}

//
//                                    ) {
//
```

The astute will be wondering why the method could not just throw an exception.
That is a very good question.

The Employee Class

We will create a Java class to store details of some employees, and a simple program to use it.

Employees have an id number, a name, and a rating.

The program needs to manage the rating, by adding or subtracting points.

The rating is always between 0 and 10.

Using a Class

The idea of using classes is they can be reused.

So writing lots of code in main is not usually the best plan.

Better is to write a class that makes use of the class being developed - it could just be used to test it.

So we will have an `EmployeeDemo` class …

IntelliJ Demo Time