Before We Begin OO ...

Some more basic Java concepts related to control flow

If with Blocks

```
public static void main(String [] args) {
  int x = 1;
  int y = 2;

  if (x == y) {
    System.out.println("x is the same as y");
  } else {
    System.out.println("x is not the same as y");
  }
}
```

Will this Compile?

```
public static void main(String [] args) {
  int x = 1;
  int y = 2;

  if (x == y)
    System.out.println(”x is the same as y”);
  else
    System.out.println(”x is not the same as y”);
}
```

What About This?

```
int x = 1;
int y = 2;

if (x == y);
  System.out.println("x is the same as y");
```

Statements and Blocks

- Statements end with ;
- Blocks are surrounded by { and }
- Blocks can be interchanged for statements in many cases.

Much more information can be found here:

`https://docs.oracle.com/javase/tutorial/java/nutsandbolts/expressions.html`

CSSE2002/7023

Programming in the Large

Week 2.1: OO Programming

In this Section

- Modelling and Practical Programming
- Types and Instances
- Invariants and Information Hiding
- Static
- Imports

Some Caveats

Object Oriented "OO" is *a* way of thinking about programs.
It is *not*:

- ...the only/best approach for all uses
  - we do focus on it in this course
- ...entirely limited to OO languages
  - but is easier in them
- ...a way to avoid basic programming skills
  - selection, repetition, functions, decomposition, ...

Perspective 1 — Modelling

In programming we often treat design as a modelling exercise.

- University: Faculties, Schools, Staff, Students, Courses, Tutes
- Public transport: Buses, Trains, Routes, Stops, Drivers, Passengers

So

1. Represent problem as a set of interacting entities.
2. Represent interactions as "messages" between entities.

It would be nice to be able to talk about entities both as groups and subgroups — "Exchange students are like Undergrad students but with the following differences."

Perspective 2 — Practical Programming

Suppose we want to represent a student
$\{$ID, name(s), program, …eye colour(?) …lift strength(?)$\}$
in our program.

- Doesn't really fit any of the standard types.
- We need a compound type.

Benefits:

- Simpler — one var is easier than twenty.
- Abstraction — Not all code needs to know what's in a student (or that it has changed).

Note: Not necessarily an explicit part of the model. The grouping may be done out of convenience (especially in Java).

Types and Instances

Types are defined by:

- State — values it can take.
- Behaviour — what can be done with those values.

For example:

Type	Values	Operations
int	1, 2, -50, . . .	+, −, *, /, %
boolean	true, false	&&, ||, !
float	1.0, 3.8, . . .	+, −, *, /, sqrt

Type can be viewed as the description of values and operations.
An “instance of a type” is an object that has a specific type.

Encapsulation

OO languages bundle behaviour into classes along with state.

- Referred to as *methods* or *member functions*.
- Data, and the code which interacts with it, are “together” (*encapsulation*).

Note: the variables which store state are referred to as *member variables* or *fields*.

CoffeeCup Example

image source

- What is required to model a coffee cup?

CoffeeCup Example

image source

- What is required to model a coffee cup?
  - It depends!

CoffeeCup Example

image source

- What is required to model a coffee cup?
    - It depends!
- If I want to know how much caffeine I am consuming from different servings of coffee?

CoffeeCup Example

image source

- What is required to model a coffee cup?
  - It depends!
- If I want to know how much caffeine I am consuming from different servings of coffee?
- I could use the amount of liquid and strength of the coffee.

CoffeeCup Example

`CoffeeCup.java` and `CoffeeCupTest.java`

Constructors

`Constructors.java`

- Classes can have multiple constructors.
- If none is explicitly declared, Java will make a default one.
- You cannot call constructors which don't exist.
  - or that you don't have access to

Constructors are how you ensure that all objects start in a valid state.

Invariants and “Interfaces”

An invariant is a condition which is “always” true.

- e.g. a student’s DOB can’t be in the future

Methods of a class should be written to preserve invariants.

Methods provide an “interface”.

But:

```
student1.dob = new Date(2258, 4, 15); // ???
```
[1]

Solution:

- Constructors should make sure objects start in a valid state.
- All methods[2] ensure they only move from one valid state to another valid state.
- *Information hiding* — block/obstruct direct access to member variables, to prevent bypassing the above.

[1] Yes, this call is deprecated

[2] called directly or triggered implicitly

Information Hiding — Access Control

Classes, member variables, and methods specify who is allowed to use them:

- `public` — Any code can use this.
- `private` — Only this class can use this.
- `protected` — (later) Only this class and subclasses can use this.
- — (later) Only code in this package can use this.

Unless you have a good reason, member variables should be private[3].

[3] Warning: this is not enough to *completely* protect your invariants

Information Hiding — Abstraction

If users of your class are calling its methods, do they *need* to know how state is represented (or how methods are implemented)?

Alternatively, can you change the internals of a class *without* breaking external code?

- Easier if they don't have direct access to internals.

Consider: `CoffeeCup2.java` and `CoffeeCupTest.java`

Last Magic — `static`

```
public class XYZ {

    public static void main(String[] args) {
        // your code here
    }

}
```

`static` members belong to class as a whole, not any particular object.

- Shared constants / flags / logging variables
- Operations which only depend on the above
    - so don't need to make an object on which the method is called
- When you don't have an object (e.g. `main`)

import

```
import java.util.List;  // use short name
                        // for one class
import java.io.*;       // use short names
                        // for whole package
```

Java's `import` statement works differently to Python's.

Python import could execute code. You can't use a class without importing it.

Java You can use a class without importing it, *if* you use its full name.

These are equivalent:

```
java.io.FileReader f;    import java.io.FileReader;
                         ...
                         FileReader f;
```

`import`

Q: If `String` is actually `java.lang.String` then why don't we need to `import` it?

A: Everything in `java.lang` is imported automatically.