Assess Yourself

You are a software developer in a swarm robotics laboratory, developing software for *heterogenous* swarms, or swarms with *multiple types* of robots.

A swarm can be an arbitrary combination of ground and aerial robots. Ground robots can be wheeled, bipedal (two legs), or spider-like (many legs). Aerial vehicles can be rotary, or winged.

Create a class design for this scenario, including appropriate use of *inheritance*, with *shared* or *common* attributes/behaviour defined in a superclass, and *specific* behaviour defined in subclasses.

Assess Yourself

Class: `Robot`

- Attributes
  - position
  - orientation
  - batteryLevel
- Methods
  - move

Assess Yourself

Class: `AerialRobot extends Robot`

- Attributes
  - altitude

Assess Yourself

Class: `RotaryRobot extends AerialRobot`

- Attributes
  - numRotors
- Methods
  - move

Assess Yourself

Class: `WingedRobot extends AerialRobot`

- Attributes
  - isPushPlane
- Methods
  - move

Assess Yourself

And so on...

SWEN20003
Object Oriented Software Development

Polymorphism and Abstract Classes

Semester 1, 2019

The Road So Far

- OOP and Java Foundations
- Classes and Objects
  - Privacy and Immutability
  - Multi-class systems
- Abstraction
  - Inheritance

Lecture Objectives

After this lecture you will be able to:

- Describe the **Object** class, and the properties inherited from it
- Describe what **upcasting** and **downcasting** are, and when they would be used
- Explain **polymorphism**, and how it is used in Java
- Describe the purpose and meaning of an **abstract** class

More on inheritance...

Object

Every class in Java implicitly inherits from the `Object` class

- All classes are of type `Object`
- All classes have a `toString` method
- All classes have an `equals` method
- ... among other (less important) things

toString

```java
public class Robot {

    private double x, y, z;

    public static void main(String[] args) {
        Robot robot = new Robot();
        System.out.println(robot);
    }
}
```

```
Robot@52e922
```

The inherited `toString` method is pretty useless, so we **override** it

toString

```
public class Robot {

    private double x, y, z;

    public static void main(String[] args) {
        Robot robot = new Robot();
        System.out.println(robot);
    }

    public String toString() {
        return String.format("Robot located at {%f, %f, %f}",
            this.x, this.y, this.z);
    }
}
```

```
"Robot located at {0, 0, 0}"
```

equals

```java
public class Robot {
    public static void main(String[] args) {
        Robot robot1 = new Robot();
        Robot robot2 = new Robot();
        System.out.println(robot1.equals(robot2));
    }
}
```

```
false
```

The inherited `equals` method is equally useless, but **overriding** is a bit more work

Assess Yourself

What do you think the *signature* would be for `equals`?

```
public class Robot {
    public boolean equals(Robot otherRobot) {
        <block of code to execute>
    }
}
```

This is actually an **overloaded** method!

Remember that `equals` is inherited from the `Object` class.

equals

```java
public boolean equals(Object other) {
    // check if references are the same
    if (this == other)
        return true;

    // check if the object exists
    if (other == null)
        return false;

    // type check before casting
    if (this.getClass() != other.getClass())
        return false;

    Robot robot = (Robot) other;
    // field comparison
    return Math.abs(this.x - robot.x) < EPS && ...;
}
```

equals

Extending equality to **subclasses**...

```
public class AerialRobot extends Robot {
    public boolean equals(Object other) {
        return super.equals(other) &&
            Math.abs(this.altitude - robot.altitude < EPS);
    }
}
```

Yay inheritance!

Useful Terms

Keyword

getClass: Returns an object of type `Class` that represents the details of the *calling object's class*.

Keyword

instanceof: An *operator* that gives `true` if an object A is an instance of the same class as object B, or a class that inherits from B.

```java
return new AerialRobot() instanceof new Robot(); // true
return new Robot() instanceof new AerialRobot(); // false
```

Useful Terms

Keyword

Upcasting: When an object of a *child* class is assigned to a variable of an *ancestor* class.

```java
Robot robot = new AerialRobot();
```

Keyword

Downcasting: When an object of an *ancestor* class is assigned to a variable of a *child* class. Only makes sense if the underlying object is **actually** of that class. Why?

```java
Robot robot = new WingedRobot();
WingedRobot plane = (WingedRobot) robot;
```

Polymorphism

Keyword

Polymorphism: The ability to use objects or methods in many different ways; roughly means "multiple forms".

Overloading	same method with various forms depending on **signature** (Ad Hoc polymorphism)
Overriding	same method with various forms depending on **class** (Subtype polymorphism)
Substitution	using subclasses in place of superclasses (Subtype polymorphism)
Generics	defining parametrised methods/classes (Parametric polymorphism, *coming soon*)

Abstract Classes

Assess Yourself

Is there anything *strange* about our `Robot` hierarchy?

- What is a `Robot`?
- If we create a `Robot` object, what does that mean?
- Does it make sense?

Abstract

How would this code work?

```java
Robot robot = new Robot();
robot.move(...)
```

It doesn't!

Some classes aren't meant to be instantiated because they aren't **well defined**.

Abstract Methods

Keyword

abstract: Defines a superclass method that is common to **all** subclasses, but has no implementation. Each subclass then provides its own implementation through **overriding**.

```
<privacy> abstract <returnType> <methodName>(<arguments>);
```

Abstract Classes

Keyword

abstract: Defines a **class** that is **incomplete**.
Classes with abstract methods **must** be abstract, but abstract classes can have no abstract methods.
Abstract classes are “general concepts”, rather than being fully realised/detailed.

```
public abstract class <ClassName> {

}
```

Abstract Classes

Keyword

Abstract Class: A class that represents common attributes and methods of its subclasses, but that is **missing** some information specific to its subclasses. Cannot be instantiated.

Keyword

Concrete Class: Any class that is not abstract, and has well-defined, specific implementations for all actions it can take.

Abstract Classes - Design 1

```
public abstract class Robot {
    public abstract void move(...);
}
```

```
public abstract class AerialRobot extends Robot {

}
```

```
public class WingedRobot extends AerialRobot {
    public void move(...) {
        <block of code to execute>
    }
}
```

Abstract Classes - Design 2

```
public abstract class Robot {
    public boolean move() {
        return canMove();
    }
}
```

```
public abstract class AerialRobot extends Robot {
    public boolean move() {
        return super.move() && activateRotors();
    }
}
```

```
public class WingedRobot extends AerialRobot {
    public boolean move(...) {
        if (super.move()) {
            setMotorSpeed(...);
            return true;
        }
    }
}
```

Back to our example...

```
Robot robot = new Robot();
```

Nope! Robot is abstract!

```
Robot robot = new AerialRobot();
```

Nope! Still abstract!

```
Robot robot = new WingedRobot();
```

Much better.

Abstract vs. Concrete

Abstract classes are identical, except:

- **May** have abstract methods
- Can't be instantiated
- Represent an **incomplete** concept, rather than a **thing** that is part of a problem

Our Powers Combine!

Let's make some magic with our new tools...

Write a program that uses the `Robot` class and its descendants to create a swarm of robots, either using input from the user, or programmatically.

If we wanted to **control** this swarm, **interact** with it, or have it operate autonomously, what abstractions might we use?

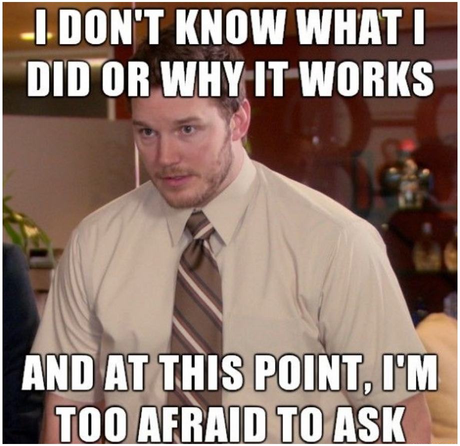
Our Powers Combine!

```
import java.util.Random;

public static void main(String[] args) {
    Robot[] swarm = new Robot[MAX_ROBOTS];

    for (int i < 0; i < MAX_ROBOTS; i++) {
        swarm[i] = createRobot();
    }
}

public Robot createRobot() {
    Random rand = new Random();

    switch(rand.nextInt(NUM_ROBOT_TYPES)) {
    case 0:
        return new WingedRobot();
    case 1:
        return new WheeledRobot();
    ...
    }
}
```

Our Powers Combine!

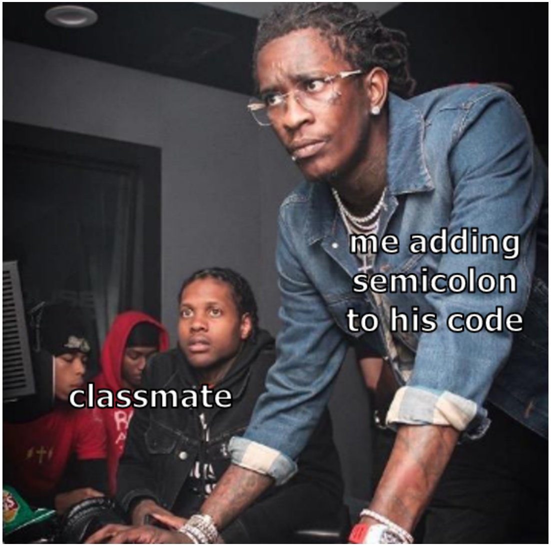
```java
public void see(Robot[] swarm) {
    for (Robot r : swarm) {
        r.see();
    }
}

public void think(Robot[] swarm) {
    for (Robot r : swarm) {
        r.think();
    }
}

public void act(Robot[] swarm) {
    for (Robot r : swarm) {
        r.act();
    }
}
```

Project 1


I DON'T KNOW WHAT I DID OR WHY IT WORKS
AND AT THIS POINT, I'M TOO AFRAID TO ASK

You have more than enough hints now.
Don't be like Chris Pratt.

Meme Submissions



Courtesy of Miro

Meme Submissions

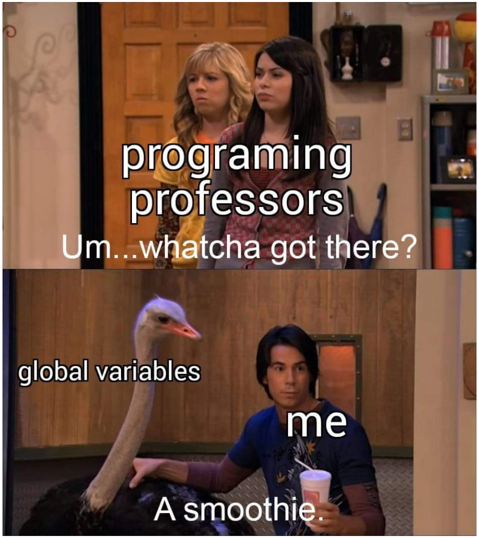
𝔅𝔩𝔞𝔦𝔫𝔢
@_Blainee

Me in class not retaining anything the professor is teaching

4:10 PM · 1/31/19 · Twitter for iPhone

Courtesy of Miro

Meme Submissions



Courtesy of Miro

Metrics

As a developer for the local zoo, your team has been asked to develop an interactive program for the student groups who often come to visit.

The system allows users to search/filter through the zoo's animals to find their favourite, where they can then read details such as the animal's favourite food, where they're found, and other fun facts.

Your teammates are handling the user interface and mechanics; you've been tasked with designing the data representation that underpins the system.

Using what you know of Object Oriented Design, design a data representation for (some of) the animals in the zoo, making sure to use *inheritance* and *abstract classes* appropriately.