



OLLSCOIL NA GAILLIMHÉ
UNIVERSITY OF GALWAY

CT2106

Object Oriented Programming



Dr. Frank Glavin
Room 404, IT Building
Frank.Glavin@UniversityofGalway.ie
School of Computer Science

University
ofGalway.ie

Contact Information

Lecturer: Dr Frank Glavin

Frank.Glavin@nuigalway.ie

Office : Room 404, Information Technology Building

Note:

The bulk of this course content was originally developed by
Dr Conor Hayes



OLLSCOIL NA GAILLIMHÉ
UNIVERSITY OF GALWAY

Lecture/Lab Times and Location

Lecture - Thursday 9 am – 10 am:
AC003, D'Arcy Thompson Lecture Theatre

Lecture - Friday: 10 am – 11 am:
IT250, Information Technology Building

Lab – Tuesday 11 pm – 1 pm:
BLE2012 Comp Suite
Arts Sci Rm 105
Block E, Ground Flr, E102

Lab – Friday 3pm – 5pm
IT106 [4BSE1 and 4BSE4]



OLLSCOIL NA GAILLIMHE
UNIVERSITY OF GALWAY

Learning Materials

- Lecture content will be provided in advance
- Lectures themselves will be in tutorial format
- You will need to bring a laptop to each lecture
- Weekly lab sessions



Attendance

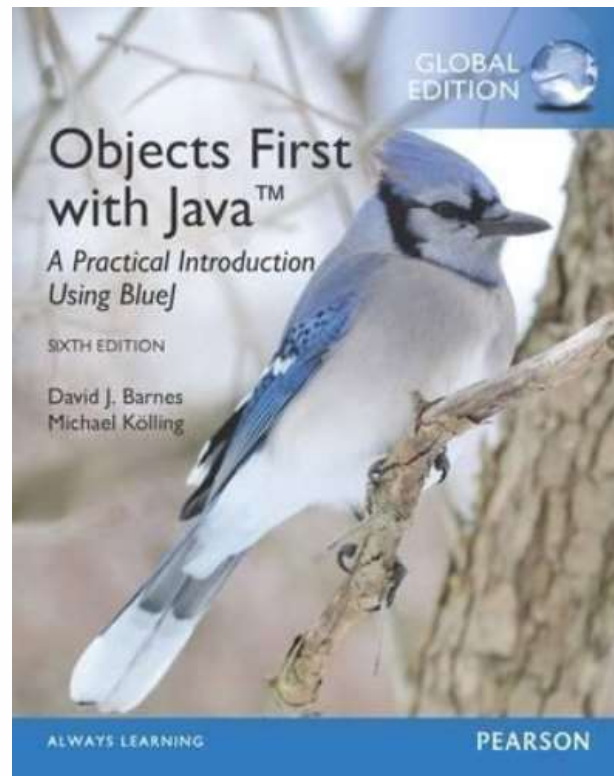
- Attendance at each lecture/tutorial will be recorded
- Attendance will be captured using the Qwickly app
- You will have time during the lecture to enter the pin



Recommended Reading

Objects First with Java:
A Practical Introduction
using BlueJ

David J. Barnes,
Michael Kölling



OLLSCOIL NA GAILLIMHÉ
UNIVERSITY OF GALWAY

Other Reading Texts

- [Think Java](http://www.greenteapress.com/thinkapjava/) by Allen B. Downey
<http://www.greenteapress.com/thinkapjava/>
- [Thinking in Java](http://www.mindview.net/Books/TIJ/) by Bruce Eckel
<http://www.mindview.net/Books/TIJ/>
- [The Java Tutorials](http://docs.oracle.com/javase/tutorial/index.html) hosted by Oracle
<http://docs.oracle.com/javase/tutorial/index.html>

Java, A Beginner's Guide, 5th Edition by Herbert Schildt

Effective Java (2nd Edition) by Joshua Bloch

Head First Java by Kathy Sierra, Bert Bates



OLLSCOIL NA GAILLIMHE
UNIVERSITY OF GALWAY

Useful Online Resources

- <https://www.geeksforgeeks.org/java/>
- <https://www.w3schools.com/java/default.asp>
- https://www.w3schools.com/java/exercise.asp?filename=exercise_syntax1
- <https://www.tutorialspoint.com/java/index.htm>
- https://www.tutorialspoint.com/java/java_online_quiz.htm



Extra Support

ComputerDISC is a Computer Programming Drop-In Support Centre for all NUI Galway students who are taking any programming/software development courses. The DISC is a free service that supports all students with their self-directed learning in computing topics at all years and levels in NUI Galway.



Room 205 in the Information Technology Building

<https://www.universityofgalway.ie/science-engineering/school-of-computer-science/currentstudents/computerdisc/>

<https://www.universityofgalway.ie/science-engineering/school-of-computer-science/currentstudents/computerdisc/timetable/>



OLLSCOIL NA GAILLIMH
UNIVERSITY OF GALWAY

Module Assessment

Assessment:

- There will be between 3 and 5 lab assignments
- Computer-based programming exam at the end of semester
- Attendance/participation at the weekly lecture tutorials
- If you should have to repeat in Autumn, your overall result is **capped** at 40%



Computer Based Exam

- In December, you will have a two-hour computer-based exam
- You will be required to solve two/three problems by programming in Java
- You will not be able to pass this exam without having developed programming competence
- Like riding a bicycle, this is not something you can learn from a book.
- You should be programming for at least two hours every week



Learning Objectives 1

Just a pass	<ul style="list-style-type: none">• Define the basic principles of OOP• List a subset of best programming practices• List the differences between OOP and procedural programming• Name the basic Java data types and demonstrate how to use these as variables• Write and compile a basic OOP program based on a given set of instructions using an IDE such as Eclipse
Quite Satisfactory	<ul style="list-style-type: none">• Create and Implement a subset of stub classes and methods so that an initial overall approach compiles• Recognise when inheritance can be used to reduce code redundancy.• Apply basic inheritance approach to solve redundancy• Implement basic software engineering best practices - such as use of methods to reduce redundancy, appropriate use of access modifiers, encapsulation• Demonstrate appropriate use of instance vs static methods/variables• Demonstrate appropriate use of getter/setter methods



Learning Objectives 2

Highly satisfactory	<ul style="list-style-type: none">• Distinguish when inheritance or an interface approach is most appropriate• Demonstrate the appropriate use of polymorphism in a coding solution• Distinguish between data structures (Arrays, ArrayLists, HashMaps, Stacks)• Recognise when to use key utility libraries in the Java language (e.g java.util. Collections) and demonstrate how to implement them
The very best understanding	<ul style="list-style-type: none">• Explain the modelling rationale for using a set of classes and methods to solve a problem description• Formulate, design and implement and test a full OO solution to a problem description• Independently recognise and apply a design pattern to solve a coding problem• Employ creative and original thinking in formulating the solution• Demonstrate a test-driven (unit-testing) approach to solving a coding problem• Assess and Compare one solution approach against another



Topics

- Classes, objects, methods
- Primitive and reference types
- Object interactions
- Arrays and collections and how to iterate
- Modelling decisions - what classes, relationships and methods to design
- Inheritance: using it to improve structure
- Polymorphism: how to use to implement the open-close principle
- Object interactions again: composition
- Java libraries
- Using Interfaces
- Good programming practice: unit testing and exception handling
- Using a design pattern to solve an OOP problem



Learning Objectives: Week 1

You should be able to:

- Describe what an Object Oriented Programming language is
- Differentiate between a class and an object
- Create a simple **class** in BlueJ and create several **objects** of that class
- Create some simple **methods** in Java



Object-oriented Programming (OOP)

What is an Object-Oriented
Programming language?



OLLSCOIL NA GAILLIMHÉ
UNIVERSITY OF GALWAY

“Hello World”

```
1 #include <stdio.h>
2
3 int main() {
4     /* my first program in C */
5
6     char hello[] = "Hello, World! \n";
7
8     printf(hello);
9
10    return 0;
11 }
```

C

```
public class Greeting
{
    public Greeting()
    {
        System.out.println("Hello World");
    }

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

Java

What are the similarities and differences between the two code snippets?



OLLSCOIL NA GAILLIMHE
UNIVERSITY OF GALWAY

Information on public static void main...

<https://www.journaldev.com/12552/public-static-void-main-string-args-java-main-method>

Definitions:

- Class

- Something from which you create objects.
 - Template

- Object

- A Java object is a self-contained component which consists of methods and properties
- E.g. in an ecommerce program, we could have customer object, item object, or book object (it will have name, ID, Price etc.)



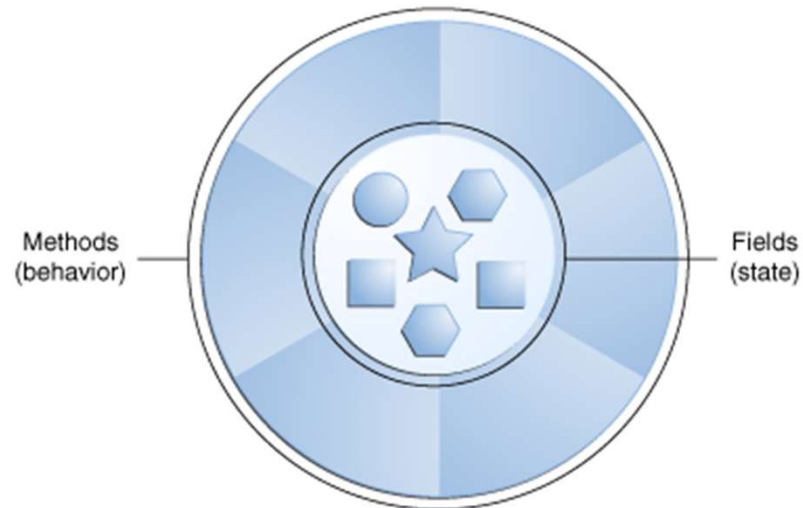
What is a class?

- A class is a type of **blueprint** or **template** from which you make objects
- The use of classes and objects are the principal differences between programming in C and programming in Java.
- However, it entails a fundamentally different way of designing your code



What is an object?

- A piece of programming code that has a **state** and has **behaviour**
- Often it represent a real thing
- It is created in code by *instantiating* a class



Bytecode

Unlike other high-level programming languages, Java code is **not** compiled into machine specific code that can be executed by a microprocessor.

Instead, Java programs are compiled into something called **bytecode**. The bytecode is input to a Java Virtual Machine (JVM), which interprets and executes the code. The JVM is usually a program itself.

The bytecode is **platform independent**. So, the JVM is specific for each platform, but the bytecode for the program remains the same across different platforms. This is a very nice feature of Java.

Of course there is always a trade off....

The main trade off is the effect it has on the execution speed.



Creating your first class

- Lets write a simple program in BlueJ
- In the lecture, you are going to
 - Create your first class
 - Create several objects of this class





OLLSCOIL NA GAILLIMHÉ
UNIVERSITY OF GALWAY

CT2106

Object Oriented Programming



Dr. Frank Glavin
Room 404, IT Building
Frank.Glavin@UniversityofGalway.ie
School of Computer Science

University
ofGalway.ie

Overview

How Java Works?

Different types of languages

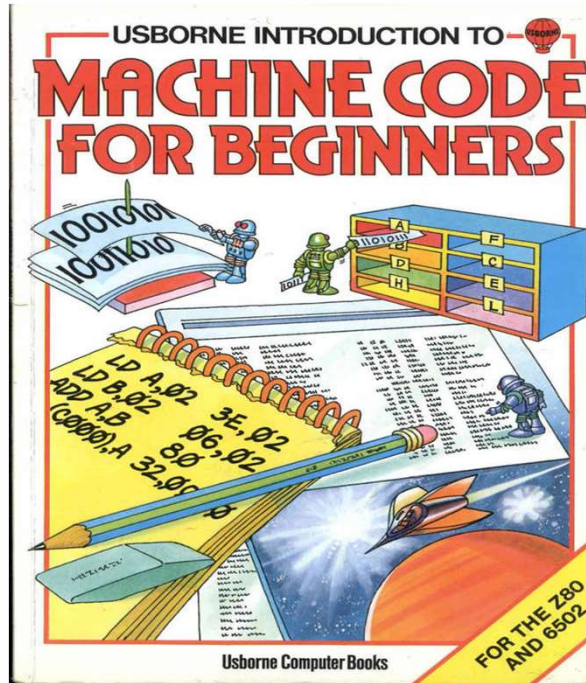
Compilation

Interpretation



OLLSCOIL NA GAILLIMH
UNIVERSITY OF GALWAY

Machine Code



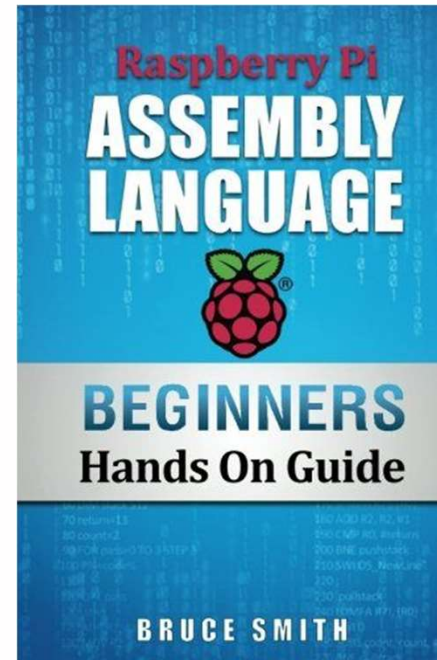
```
ba 0c 01
b4 09
cd 21
b8 00 4c
cd 21
48 65 6c 6c 6f 2c
20 57 6f 72 6c 64
21 0d 0a 24
```



OLLSCOIL NA GAILLIMHIE
UNIVERSITY OF GALWAY

Assembly

```
mov dx, 010ch
mov ah, 09
int 21h
mov ax, 4c00h
int 21h
db 'Hello, World!', '$'
```



OLLSCOIL NA GAILLIMHÉ
UNIVERSITY OF GALWAY

Java

```
public class Greeting
{
    public Greeting()
    {
        System.out.println("Hello World");
    }

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

C

```
1 #include <stdio.h>
2
3 int main() {
4     /* my first program in C */
5
6     char hello[] = "Hello, World! \n";
7
8     printf(hello);
9
10    return 0;
11 }
```

Assembly

```
mov dx, 010ch
mov ah, 09
int 21h
mov ax, 4c00h
int 21h
db 'Hello, World!', '$'
```

Machine Code

```
ba 0c 01
b4 09
cd 21
b8 00 4c
cd 21
48 65 6c 6c 6f 2c
20 57 6f 72 6c 64
21 0d 0a 24
```



High-level vs Low-level

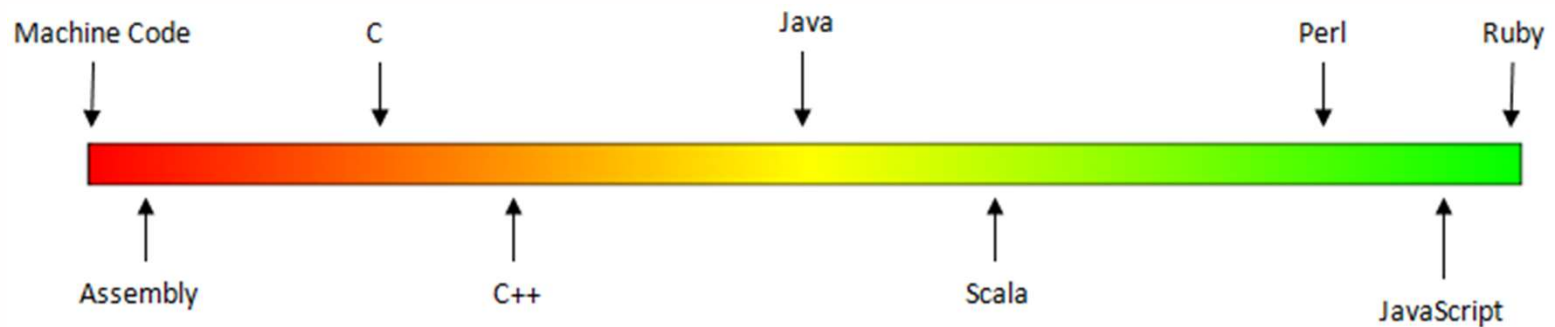
- Both Java and C are **high-level languages** and assembly is a **low-level language**
- What does that mean?



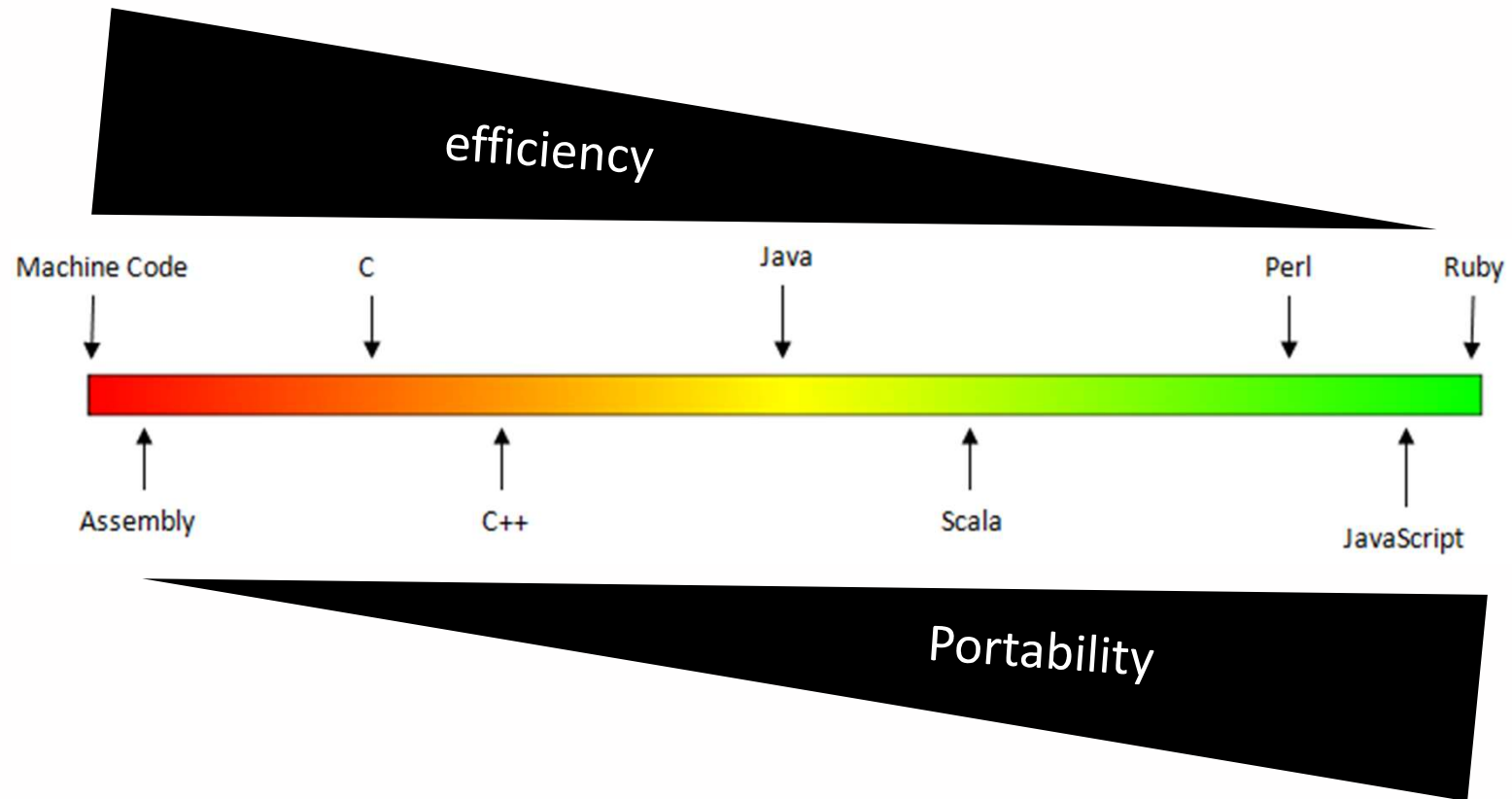
High Level Language

'High level' is a *relative* term - the level of abstraction above a **low level language**

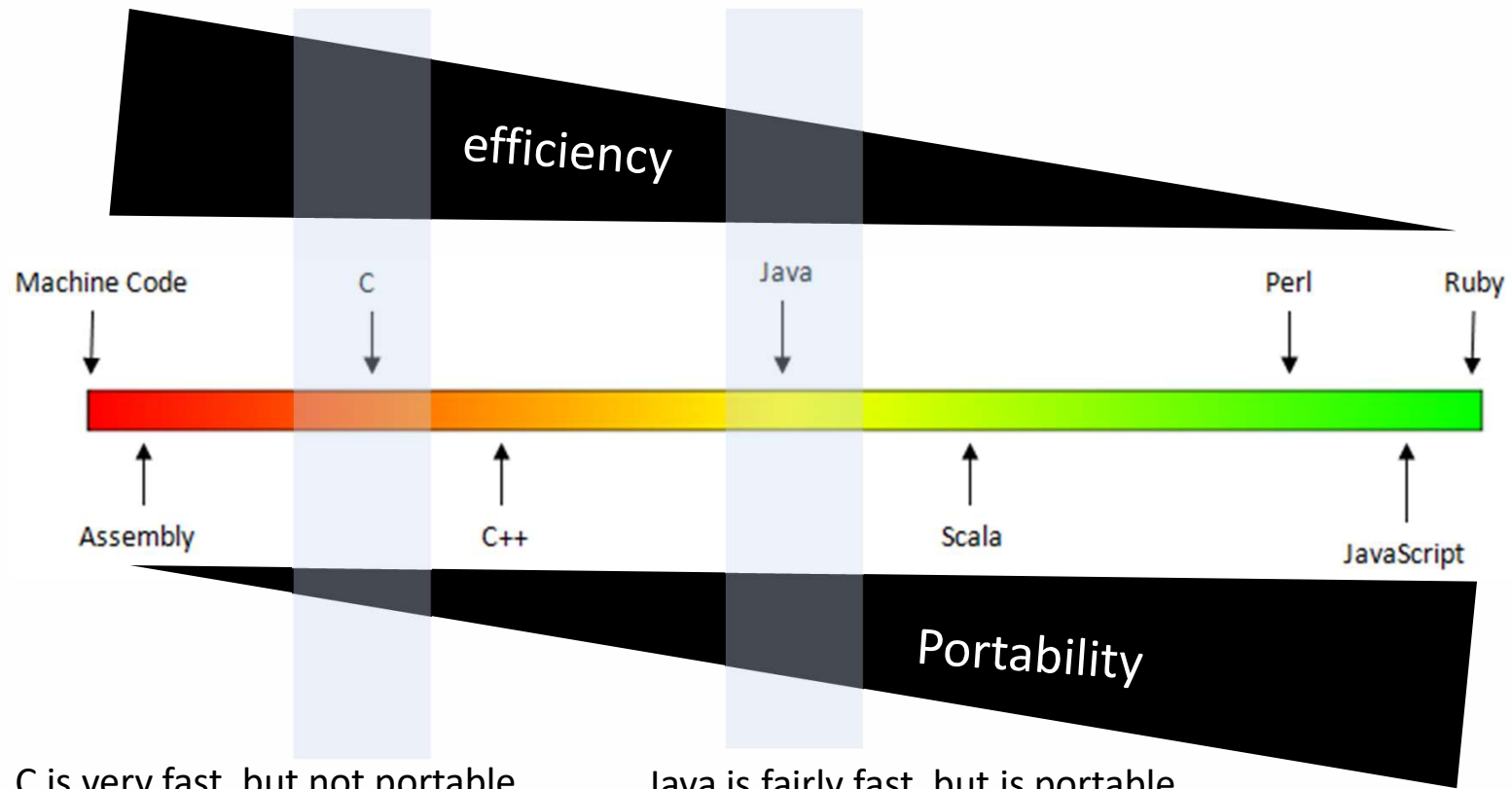
A **low level language** has little or no abstraction over the machine code of a particular processor.



High-level vs Low-level Language



High Level vs Low level Language



C is very fast, but not portable

Java is fairly fast, but is portable



OLLSCOIL NA GAILLIMHĒ
UNIVERSITY OF GALWAY

High Level Programming Languages

- Easier to program in a high- level language
- Syntax can be understood by people
- Program takes less time to write, shorter and easier to read, more likely to be correct.
- Portable – they can be run on different kinds of computers



Translating your code so that it runs

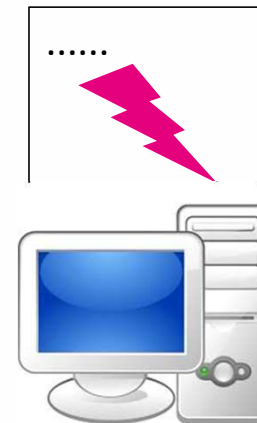
- Unless you are writing machine code (!) – your code has to be translated into machine code to run on your computer

Code

```
1 #include <stdio.h>
2
3 int main() {
4     /* my first program in C */
5
6     char hello[] = "Hello, World! \n";
7
8     printf(hello);
9
10    return 0;
11 }
```

Translation

Machine code



Two Types of Translation

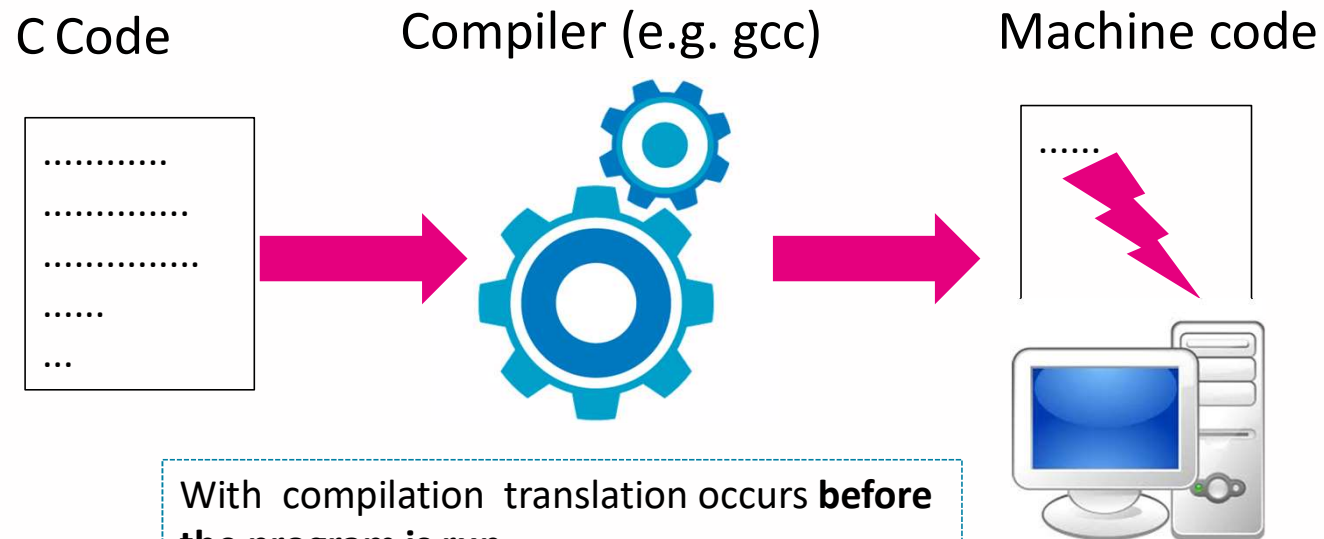
Compilation

Interpretation



C is a compiled language

- A compiler is a program that takes human readable source code and translates it **in one go** into machine code using a Compiler



With compilation translation occurs **before the program is run**



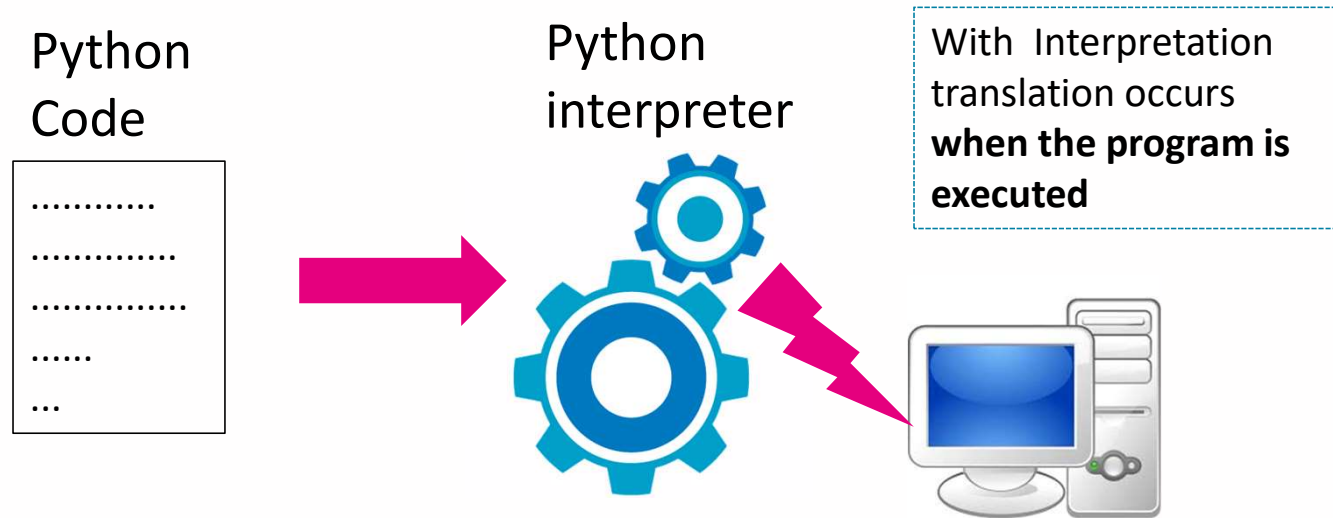
Compilation

- A compiler translates source code **in one go** into machine code for a particular machine
- However, **the machine code generated is not portable**
- You have to compile the code again if you want it to run on a different type of machine.
- However, the generated code typically executes very efficiently



Interpretation

- The second type of translation approach
- Code is **translated on-the-fly at runtime** into commands that can be executed on the machine.



Compilation vs Interpretation

Compilation

- A compiler translates source code **in one go** into machine code before the programme is run
- Typically, translating to native machine code means **very efficient run-time speed**
- For big projects, compile time can be slow

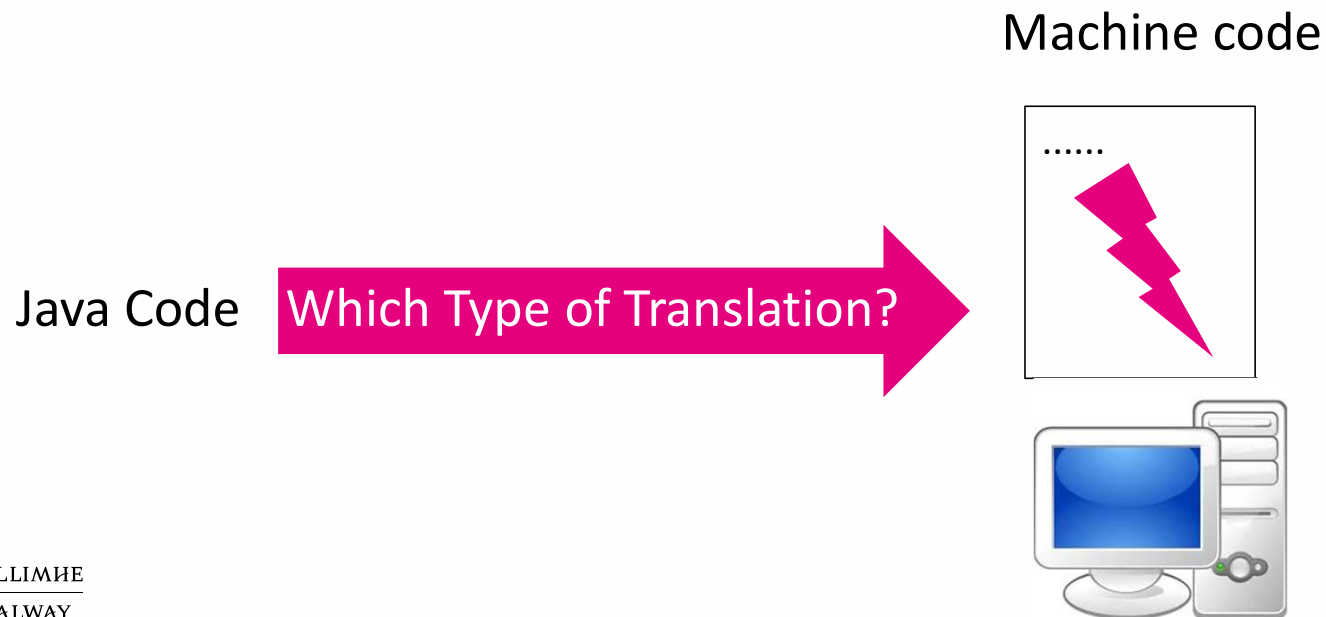
Interpretation

- Code read and executed by another program (the interpreter) when the program is run
- This makes the code **portable** (as long as there is an interpreter)
- Typically, slower to run as each statement has to be interpreted into machine code **on-the-fly**
- Greater chance of run-time errors



Translating Java Code

It is important to understand how and why Java does this differently



Java's Design Goals include:

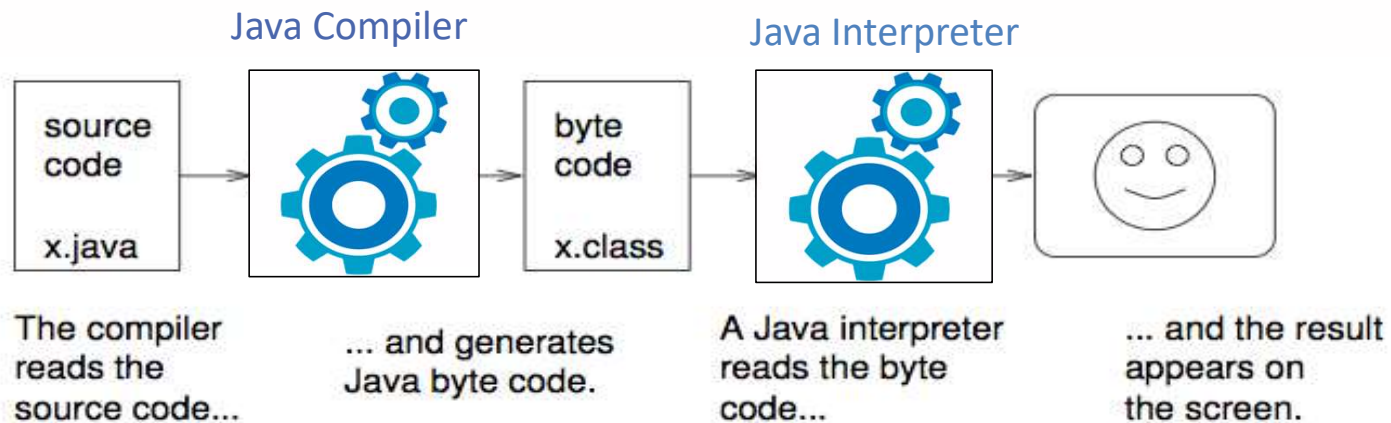
- **Portability** (typically interpreted languages)
- **High Performance** (typically compiled languages)
- How does Java achieve both?



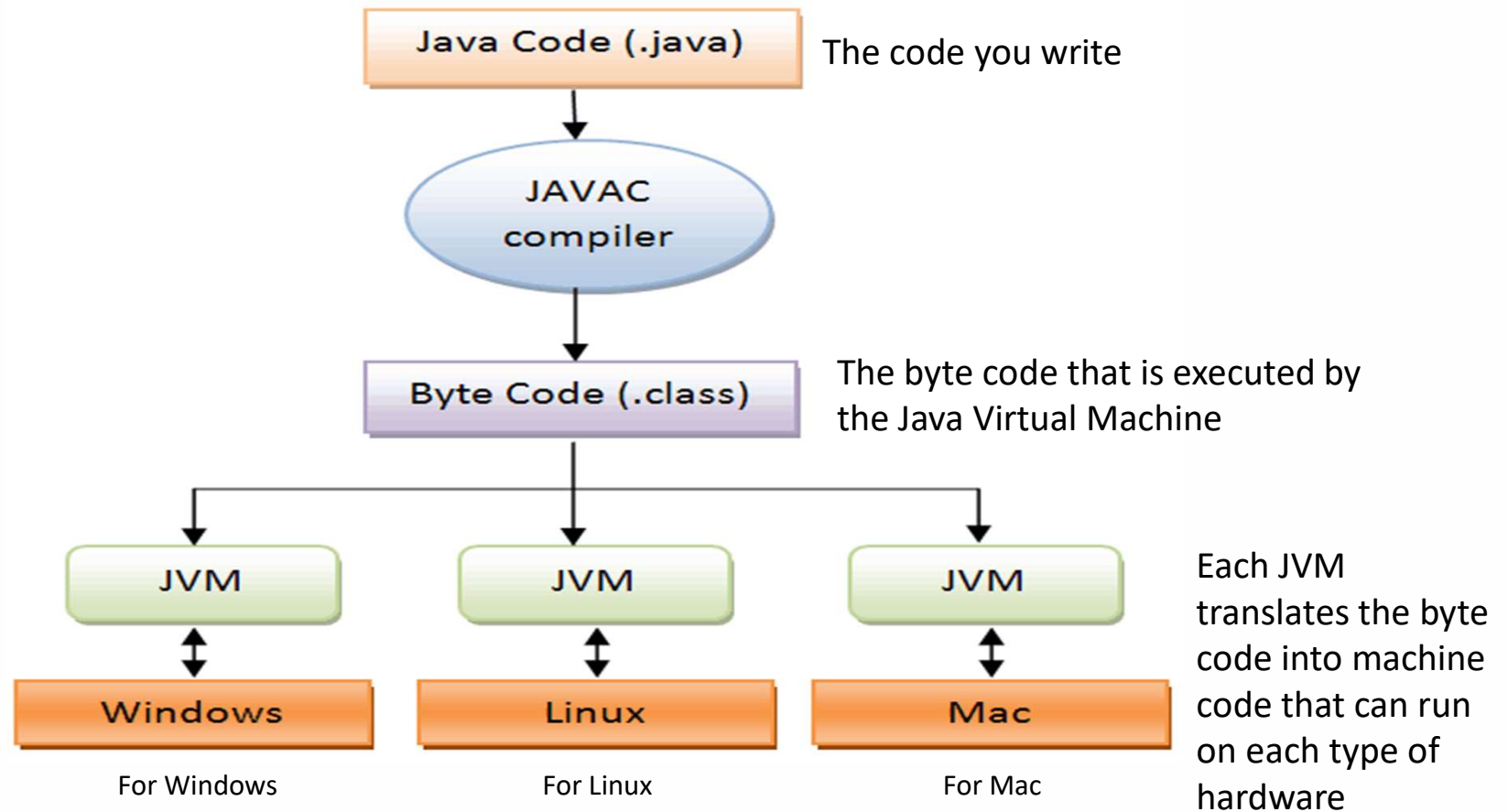
Java Translation

Java is typically both *compiled* and *interpreted*.

1. Java is **compiled** to *Byte Code* – *an intermediate language* which is portable
2. Then a Java **interpreter** reads and executes the Byte Code



Java Architecture



Java Virtual Machine (JVM)

- JVM is a piece of software not hardware
- A virtual computer on which Java byte code is executed
- Oracle provide a JVM abstract specification and a concrete implementation for each operating system type (e.g. Windows, OSX, Linux)
- There are multiple other specialised JVMs that all run
- See: https://en.wikipedia.org/wiki/List_of_Java_virtual_machines#ActiveJava








Java Runtime Environment (JRE)

- JRE contains the JVM and all libraries required to run the Java Program



What happens when you compile code?

- Open BlueJ
- Compile an existing or new project
- Go to your Project Folder
- You will see 5 **files**

Name	Date modified	Type	Size
 GreetingAll.class	19/09/2018 11:45	CLASS File	1 KB
 GreetingAll.ctxt	19/09/2018 11:45	CTXT File	1 KB
 GreetingAll.java	13/09/2018 09:50	Java Source File	1 KB
 package.bluej	19/09/2018 11:45	BlueJ Project File	1 KB
 README.TXT	10/09/2018 17:04	Text Document	1 KB



Summary of How Java Works

- Java is a high-level language.
- Its source code is compiled to intermediate level bytecode
- Bytecode is executed on the Java Virtual Machine



Learning exercise

In Blue J:

Create a Bicycle class and a Car class

Each Bicycle object should its own speed and gear (.e.g. 1st, 2nd, 3rd etc) state

What type of variable in Java could be used to represent speed and gear (look it up on the Web)?

Create **setSpeed** and **setGear** method that can set the speed /gear state of a bicycle and a car object **and print out the current speed of each**

Then Create 3 Bicycle and 3 Car objects

Using the methods above set and print different speed and gear values for each





OLLSCOIL NA GAILLIMHE
UNIVERSITY OF GALWAY

CT2106

Object Oriented Programming



Dr. Frank Glavin
Room 404, IT Building
Frank.Glavin@UniversityofGalway.ie
School of Computer Science

University
ofGalway.ie

Last Lecture - First Java Code

- In the last session, you wrote your first class and created several objects from it
- You were introduced to the notion of **state**
 - **Every object has its own state**
- An object's state is generally defined by the values it holds
- Multiple objects can be created from a single class. Each object can have its own state.



Topics

By the end of this lecture you will be able to implement the following in Java:

- Correct class and method structure
- Define and initialise an int variable
- Use accessor and mutator methods
- Explain the concept of encapsulation
- Print out the object state
- Use the Java conditional statement (if else)



Today's Learning exercise

In Blue J:

- Create a Bicycle class and a Car class
- Each Bicycle object should its own speed, gear and cadence (e.g. 1st, 2nd, 3rd etc) state
- What type of variable in Java could be used to represent speed, gear and cadence (look it up on the Web)?
- Create **setSpeed**, **setGear** and **setCadence** method that can set the speed /gear state of a bicycle and a car object **and print out the current speed of each**
- Then Create 3 Bicycle and 3 Car objects
- Using the methods above set and print different speed, gear and cadence values for each



Class Structure:

Every class has the following structure

```
public class ClassName
{
    Fields
    Constructors
    Methods
}
```



Fields

- Fields store values for an object.
- They are also known as *instance variables*.
- Fields define the state of an object.
- Use *Inspect* in BlueJ to view the state.
- Some values change often.
- Some change rarely (or not at all).

```
public class Bicycle
{
    private int speed;
    private int gear;
    private int cadence;

    Further details omitted.
}
```

visibility modifier type variable name

private int speed;



Data Type: int

Java Primitive Types

Type	Size	Range	Default
boolean	1 bit	true or false	false
byte	8 bits	[-128, 127]	0
short	16 bits	[-32,768, 32,767]	0
char	16 bits	['\u0000', '\uffff'] or [0, 65535]	'\u0000'
int	32 bits	[-2,147,483,648 to 2,147,483,647]	0
long	64 bits	$[-2^{63}, 2^{63}-1]$	0
float	32 bits	32-bit IEEE 754 floating-point	0.0
double	64 bits	64-bit IEEE 754 floating-point	0.0



Principle 1 of OOP: Encapsulation

In encapsulation, the variables of a class will be hidden from other classes and can be accessed only through the methods of their current class, therefore it is also known as **data hiding**.

- Why?
- Basic OOP philosophy: **each object is responsible for its own data**
- This allows an object to have much greater **control**
 - Which data is available to be viewed externally
 - How external objects may change (mutate) the object's state



Encapsulation Type: Private

- Making the fields **private** encapsulates their values inside each object
- No external class or object can access them.

```
public class Bicycle
{
    private int speed;
    private int gear;
    private int cadence;

    Further details omitted.
}
```



Constructors (1)

- Initialize an object.
- Have the same name as their class.
- Close association with the fields:
 - Initial values stored into the fields.
 - Parameter values often used for these.

```
public Bicycle(int spd, int gr, int cad)
{
    speed = spd;
    gear = gr;
    cadence = cad;
}
```



Constructors (2)

- If input parameter variables have the **same name** as your fields
- Then you must use the **this** keyword to distinguish between the two
- this = “belonging to this object”

```
public Bicycle(int speed, int gear, int cadence)
{
    this.speed = speed;
    this.gear = gear;
    this.cadence = cadence;
}
```



Choosing Variable Names

- There is a lot of freedom over choice of names. Use it wisely!
- Choose expressive names to make code easier to understand:
 - price, amount, name, age, etc.
- Avoid single-letter or cryptic names:
 - w, t5, xyz123



Methods

- Methods implement the *behaviour* of an object.
- Methods have a consistent structure comprised of a *header* and a *body*.
- **Accessor methods** provide information about the state of an object.
- **Mutator methods** alter the state of an object.
- Other sorts of methods accomplish a variety of tasks.

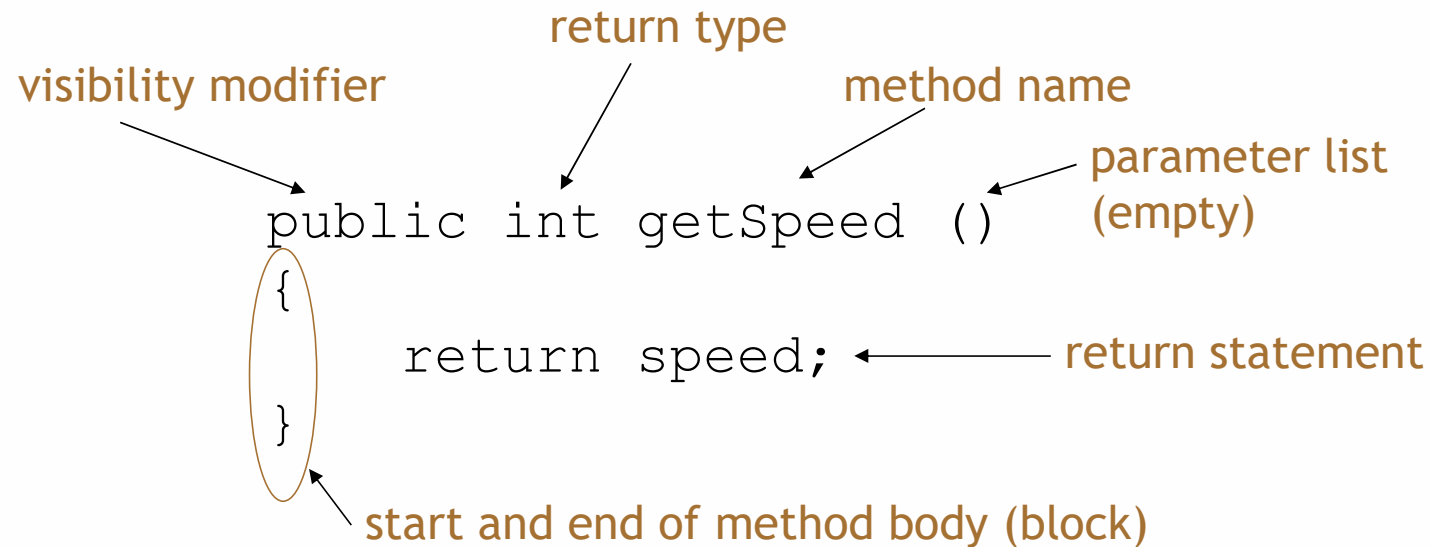


Method structure

- The header:
 - **public int getSpeed ()**
- The header tells us:
 - the *visibility* to objects of other classes;
 - whether the method *returns a result*;
 - the *name* of the method;
 - whether the method takes *parameters*.
- The body encloses the method's *statements*.



Accessor (**get**) methods

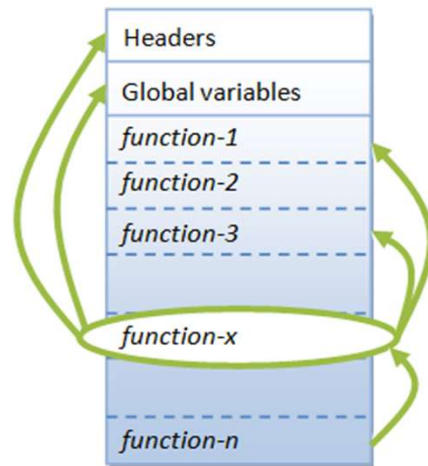


Accessor methods

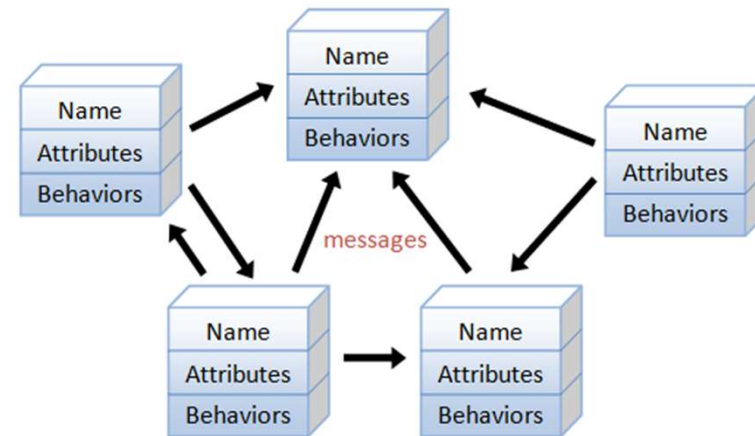
- An accessor method always has a return type that is not **void**.
- An accessor method returns a value (*result*) of the type given in the header.
- The method will contain a **return** statement to return the value.
- NB: Returning is not printing!



C vs. Java



A function (in C) is not well-encapsulated



An object-oriented program consists of many well-encapsulated objects and interacting with each other by sending messages

- Unlike a C program, an OOP program **will not** have a pool of global variables that each method can access
- Instead, **each object has its own data** – and other objects rely upon the *accessor* methods of the object to access the data




```

public class Bicycle {

    private int cadence;
    private int speed;
    private int gear;

    public int getCadence() {
        return cadence;
    }

    public void setCadence(int newValue) {
        cadence = newValue;
    }

    public int getGear() {
        return gear;
    }

    public void setGear(int newValue) {
        gear = newValue;
    }

    public int getSpeed() {
        return speed;
    }

    ...
}

```

- The instance variables (or fields) are declared **private**
- Cannot be accessed directly
- accessor/mutator methods used to access the data
- These are often called **getter/setter** methods



Test:

```
public class Bicycle
{
private speed;

public Bicycle()
{
    speed = 300
}

public int getSpeed
{
    return Speed;
}
```

What is wrong here?
(there are **five** errors!)

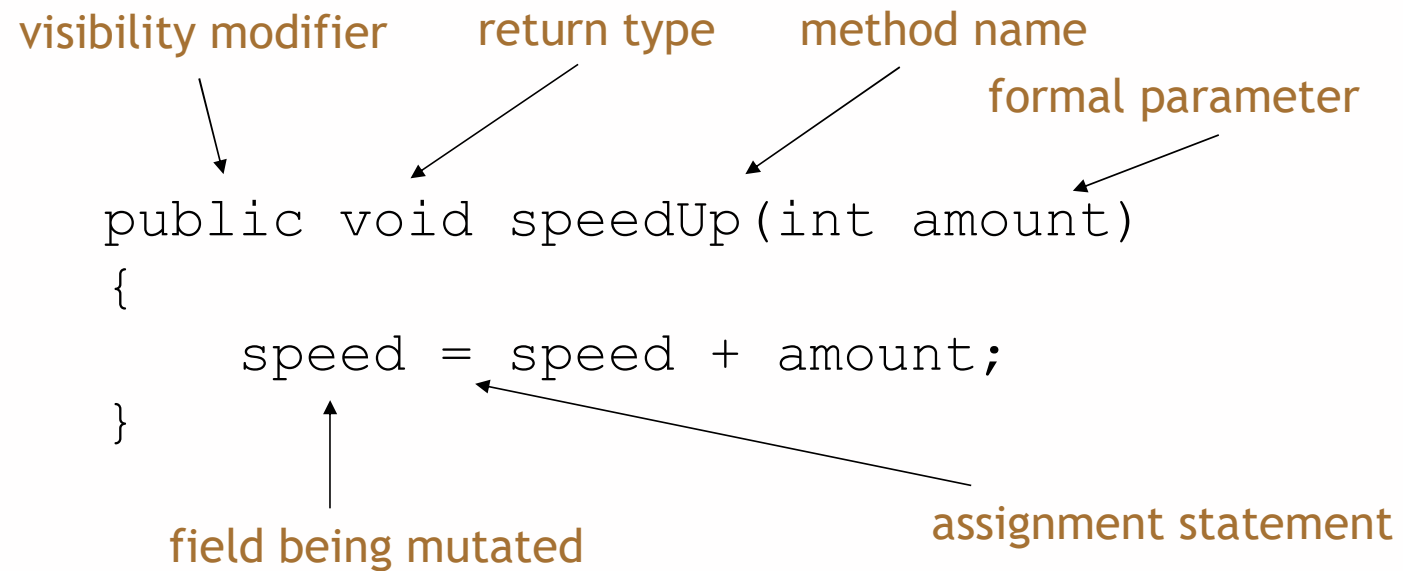


Mutator Methods (1)

- Have a similar method structure: header and body.
- Used to *mutate* (i.e., change) an object's state.
- Achieved through changing the value of one or more fields.
They typically contain one or more assignment statements.
Often receive parameters.



Mutator Methods (2)



Mutator Methods: 'set'

- Each field may have a dedicated **set** mutator method.
- These have a simple, distinctive form:
 - void** return type
 - method name related to the field name
 - single formal parameter, with the same type as the type of the field
 - a single assignment statement



Mutator Methods: 'set'

- A typical 'set' method

```
public void setGear(int number)
{
    gear = number;
}
```

- We can easily infer that gear is a field of type 'int',
 - private int gear;



Protective Mutators

- A set method does not have to always assign unconditionally to the field.
- The parameter may be checked for validity and rejected if inappropriate.
- Mutators thereby protect fields.
- Mutators support *encapsulation*.



Printing From Methods

```
public void printState()
{
    // Simulates output from a bike computer.
    System.out.println("#####");
    System.out.println("# Speed: " + speed);
    System.out.println("# Gear : " + gear);
    System.out.println("# Cadence: " + cadence);
    System.out.println("#####");
    System.out.println();
}
```



Printing From Methods 2

```
public void printState()
{
    // Simulates output from a bike computer.
    System.out.println("#####");
    System.out.printf("# Speed: %d \n ", speed);
    System.out.printf("# Gear : %d \n ", gear);
    System.out.printf("# Cadence: %d \n", cadence);
    System.out.println("#####");
    System.out.println();
}
```



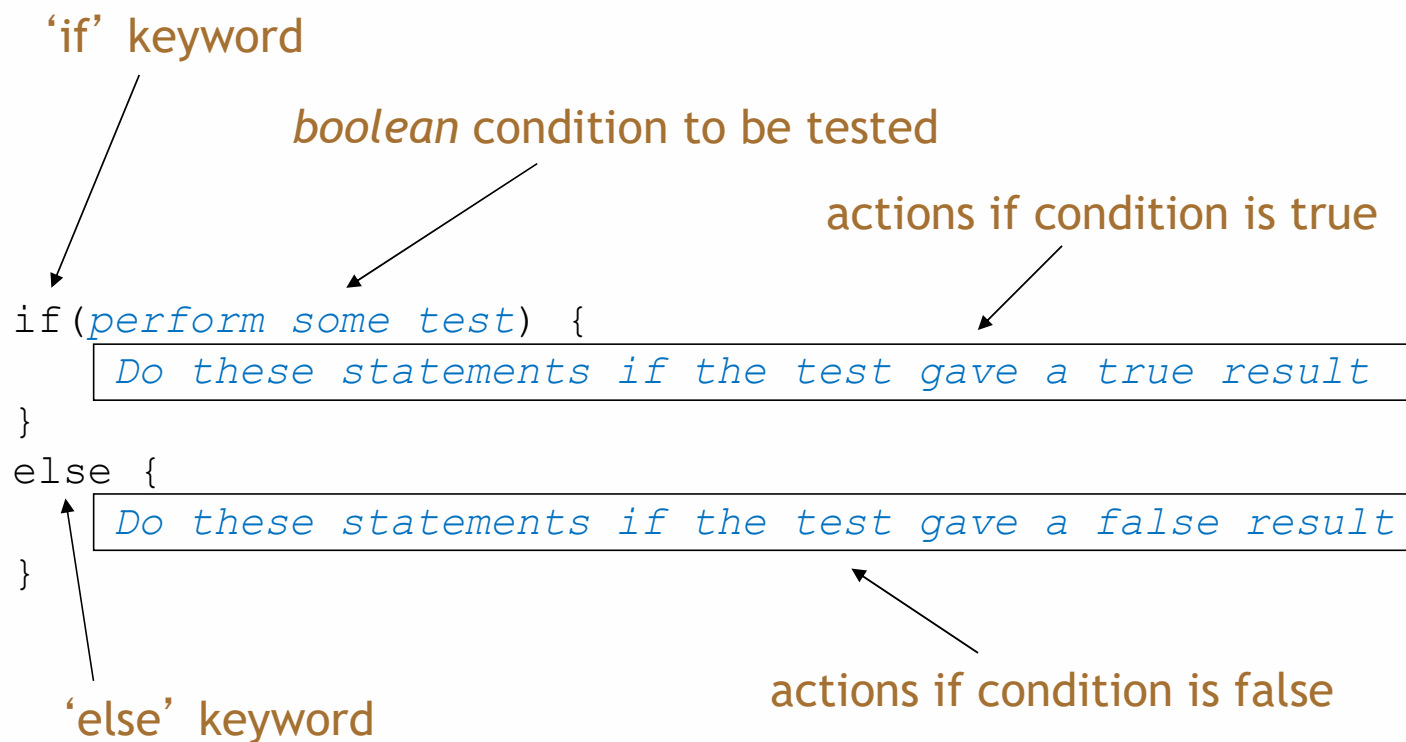
Conditional Statement

```
if(I have enough money left) {  
    I will go out for a meal;  
} else {  
    I will stay home and watch a movie;  
}
```

- It has the same format that you have seen in C



Making choices in Java



Protecting a Field (1)

```
public void setGear(int gearing)
{
    if(gearing <= 18) {
        gear = gearing;
    }
    else {
        System.out.println(
            "Exceeds maximum gear ratio.
            Gear not set");
    }
}
```

This conditional statement avoids an inappropriate action. It protects the gear field from too large values



Protecting a Field (2)

```
public void setGear(int gearing)
{
    if(gearing >= 1 && gearing <= 18) {
        gear = gearing;
    }
    else {
        System.out.println(
            "gear input value not in the
            correct range");
    }
}
```

This conditional statement avoids an inappropriate action. It protects the gear field from too large AND too small values



Summary

- You have encountered some of the fundamental ideas in OOP
- A class has fields, a constructor(s) and methods
- Encapsulation - each object's data (fields) is protected by its accessor/mutator methods
- If you want to access/change an object's state, you must use its accessor/mutator methods
- The use of the 'private' keyword prevents external access to an object's data





OLLSCOIL NA GAILLIMHÉ
UNIVERSITY OF GALWAY

CT2106

Object Oriented Programming



Dr. Frank Glavin
Room 404, IT Building
Frank.Glavin@UniversityofGalway.ie
School of Computer Science

University
ofGalway.ie

Summary of Last Two Lectures

- A class has fields, a constructors and methods
- Encapsulation - each object's data (fields) is protected by its accessor/mutator methods
- If you want to access/change an object's state, you must use its accessor/mutator methods
- The use of the 'private' key word prevents external access to an object's data
- Java is both compiled and Interpreted
- Java uses JVM to execute the same code on multiple platforms/machines



Today's Lecture

- How to implement a scenario?
- An object can be composed of other objects
- Objects can call each other's methods
- Java uses Reference types as well as primitive types
- What to watch out for in Integer division
- To use double and boolean primitive values
- To use conditional statements



An Example Problem to Solve/Implement

We wish to be able to create several Car (objects)

Each car object has an Engine

Each Engine has the following properties

kpg (kilometers per gallon)

fuel (amount of fuel in the tank)

Each Car has a **totalDistance** (travelled)



Problem

Each Car should have a **move method** specifying the distance to be travelled

You may call this method as often as you wish, and the car will print out

- Total distance travelled so far
- Remaining fuel
- Estimated distance left to travel

If you are out of fuel, the car will notify you



How to Start

Firstly, identify the classes

Code up the basic classes

Remember each class should have

- Fields

- At least one constructor

- Methods



Linking classes

Each Car object “**has a**”/ “**has an**” Engine

In OOP terms, this means that a Car object relies upon the service of an Engine object



Is-a vs has-a relationships

- Two fundamental relationships between classes in OOP
 - **has-a** (or composition)
 - **is-a** (or inheritance) : we'll encounter this later
- A RacingBike is-a type of Bicycle (Inheritance)
- A RacingBike has-a Wheel (Composition)



Representing **has-a** relationships

- **has-a** relationship denotes **composition**
- One object is **composed** of another and relies upon its services for its own functionality
- A Vehicle **has-a(n)** Engine; A Bicycle has a wheel



Representing **has-a** relationships

- In OOP class diagrams a diamond shape like this indicates a composition or has-a relationship



- This class diagram tell us that a Vehicle object is composed of a single Engine object



Realising composition in Java

- To realise a has-a relationship you have to create a link between the participant classes
- You do this using a new type of variable type: **a reference variable type**
- The reference declaration is in the **owner** class
- In our example, the Car class will have reference variable that points to an Engine object



```

public class Bicycle
{
    // instance variables - replace the example below
    private int speed;
    private int gear;
    private int cadence;
    private Wheel front;
    private Wheel back;

    /**
     * Constructor for objects of class Bicycle
     */
    public Bicycle(int speed, int gear, int cadence)
    {
        // initialise instance variables
        this.speed = speed;
        this.gear = gear;
        this.cadence = cadence;
        front = new Wheel(5);
        back = new Wheel (5);
    }
}

```

Two reference variable of type Wheel are declared

The variables are initialised in the constructor



Wheel Class

```
public class Wheel
{
    // instance variables - replace the example below with
    private int radius;

    /**
     * Constructor for objects of class Wheel
     */
    public Wheel(int radius)
    {
        // initialise instance variables
        this.radius = radius;
    }
}
```



Following this example, you can create a link between Car and Engine



OLLSCOIL NA GAILLIMHÉ
UNIVERSITY OF GALWAY

Information Required

- What information does the Car object require from Engine object?
 - “Each car should have a **move method** specifying the distance to be travelled”
- You may call this method as often as you wish and the Car will print out:
 - Total distance travelled so far
 - Remaining fuel
 - Estimated distance left to travel

“If you are out of fuel, the car will notify you”



Objects Communicating

- What information does the Car object require from Engine object?
-
- We know this
 - Engine object has:
 - ❖ Fuel amount
 - ❖ kpg (the amount of fuel used per distance)
- Car object has
 - The distance amount
 - The total distance travelled amount
 - A move method



Car to Engine

- Car has no information about fuel levels
- It requires Engine to give it that

Engine to Car

- Engine has no information about distance
- It requires Car to give it this (so that it can calculate fuel consumption)



go(int distance) method in Engine class

```
/**
 * go method of the engine calculates the amount of fuel needed to go
 * the distance required. It updates the fuel variable based on this calculation.
 * It returns false if the updated fuel calculation is less than zero.
 * This is a rough and ready way to determine if the fuel level can accomodate the distance required.
 * Can you do better ? For example, if there was fuel for 5 km, but the distance variable was 10km
 * perhaps this method should return the distance that could be travelled, rather
 * than returning false.
 *
 * @param distance : the distance required to travel
 * @return true or false based on whether it is possible or not
 */
public boolean go(int distance)
{
    fuel = fuel - distance/kpg; // integer division problem here. Can you spot it?
    if(fuel >=0){
        return true;
    }

    return false;
}
```



setFuel(int fuel) from the **Car** class

```
public void setFuel(int fuel){  
    engine.setFuel(fuel);  
}
```



move(int distance) from the Car class

```
/**
 * The move method is called whenever a Car object is required to move
 *
 * @param distance : the distance the car wishes to move
 * @return boolean: true or false based on whether the car moved or not
 */
public boolean move(int distance)
{
    boolean moved = engine.go(distance); //checks to see if engine will allow this distance

    if(moved){
        totalDistance +=distance; //updates distance travelled
    }

    return moved;
}
```



First Assignment

- Based on this example sand will be posted later today.
- It will be due next Friday.





OLLSCOIL NA GAILLIMHE
UNIVERSITY OF GALWAY

CT2106

Object Oriented Programming



Dr. Frank Glavin
Room 404, IT Building
Frank.Glavin@UniversityofGalway.ie
School of Computer Science

University
ofGalway.ie

Summary of Last Two Lectures

- A class has fields, a constructors and methods
- Encapsulation - each object's data (fields) is protected by its accessor/mutator methods
- If you want to access/change an object's state, you must use its accessor/mutator methods
- The use of the 'private' key word prevents external access to an object's data
- Java is both compiled and Interpreted
- Java uses JVM to execute the same code on multiple platforms/machines



Today's Lecture

- How to implement a scenario?
- An object can be composed of other objects
- Objects can call each other's methods
- Java uses Reference types as well as primitive types
- What to watch out for in Integer division
- To use double and boolean primitive values
- To use conditional statements



An Example Problem to Solve/Implement

We wish to be able to create several Car (objects)

Each car object has an Engine

Each Engine has the following properties

kpg (kilometers per gallon)

fuel (amount of fuel in the tank)

Each Car has a **totalDistance** (travelled)



Problem

Each Car should have a **move method** specifying the distance to be travelled

You may call this method as often as you wish, and the car will print out

- Total distance travelled so far
- Remaining fuel
- Estimated distance left to travel

If you are out of fuel, the car will notify you



How to Start

Firstly, identify the classes

Code up the basic classes

Remember each class should have

- Fields

- At least one constructor

- Methods



Linking classes

Each Car object “**has a**”/ “**has an**” Engine

In OOP terms, this means that a Car object relies upon the service of an Engine object



Is-a vs has-a relationships

- Two fundamental relationships between classes in OOP
 - **has-a** (or composition)
 - **is-a** (or inheritance) : we'll encounter this later
- A RacingBike is-a type of Bicycle (Inheritance)
- A RacingBike has-a Wheel (Composition)



Representing **has-a** relationships

- **has-a** relationship denotes **composition**
- One object is **composed** of another and relies upon its services for its own functionality
- A Vehicle **has-a(n)** Engine; A Bicycle has a wheel



Representing **has-a** relationships

- In OOP class diagrams a diamond shape like this indicates a composition or has-a relationship



- This class diagram tell us that a Vehicle object is composed of a single Engine object



Realising composition in Java

- To realise a has-a relationship you have to create a link between the participant classes
- You do this using a new type of variable type: **a reference variable type**
- The reference declaration is in the **owner** class
- In our example, the Car class will have reference variable that points to an Engine object



```

public class Bicycle
{
    // instance variables - replace the example below
    private int speed;
    private int gear;
    private int cadence;
    private Wheel front;
    private Wheel back;

    /**
     * Constructor for objects of class Bicycle
     */
    public Bicycle(int speed, int gear, int cadence)
    {
        // initialise instance variables
        this.speed = speed;
        this.gear = gear;
        this.cadence = cadence;
        front = new Wheel(5);
        back = new Wheel (5);
    }
}

```

Two reference variable of type Wheel are declared

The variables are initialised in the constructor



Wheel Class

```
public class Wheel
{
    // instance variables - replace the example below with
    private int radius;

    /**
     * Constructor for objects of class Wheel
     */
    public Wheel(int radius)
    {
        // initialise instance variables
        this.radius = radius;
    }
}
```



Following this example, you can create a link between Car and Engine



OLLSCOIL NA GAILLIMHÉ
UNIVERSITY OF GALWAY

Information Required

- What information does the Car object require from Engine object?
 - “Each car should have a **move method** specifying the distance to be travelled”
- You may call this method as often as you wish and the Car will print out:
 - Total distance travelled so far
 - Remaining fuel
 - Estimated distance left to travel

“If you are out of fuel, the car will notify you”



Objects Communicating

- What information does the Car object require from Engine object?
-
- We know this
 - Engine object has:
 - ❖ Fuel amount
 - ❖ kpg (the amount of fuel used per distance)
- Car object has
 - The distance amount
 - The total distance travelled amount
 - A move method



Car to Engine

- Car has no information about fuel levels
- It requires Engine to give it that

Engine to Car

- Engine has no information about distance
- It requires Car to give it this (so that it can calculate fuel consumption)



go(int distance) method in Engine class

```
/**
 * go method of the engine calculates the amount of fuel needed to go
 * the distance required. It updates the fuel variable based on this calculation.
 * It returns false if the updated fuel calculation is less than zero.
 * This is a rough and ready way to determine if the fuel level can accomodate the distance required.
 * Can you do better ? For example, if there was fuel for 5 km, but the distance variable was 10km
 * perhaps this method should return the distance that could be travelled, rather
 * than returning false.
 *
 * @param distance : the distance required to travel
 * @return true or false based on whether it is possible or not
 */
public boolean go(int distance)
{
    fuel = fuel - distance/kpg; // integer division problem here. Can you spot it?
    if(fuel >=0){
        return true;
    }

    return false;
}
```



setFuel(int fuel) from the **Car** class

```
public void setFuel(int fuel){  
    engine.setFuel(fuel);  
}
```



move(int distance) from the Car class

```
/**
 * The move method is called whenever a Car object is required to move
 *
 * @param distance : the distance the car wishes to move
 * @return boolean: true or false based on whether the car moved or not
 */
public boolean move(int distance)
{
    boolean moved = engine.go(distance); //checks to see if engine will allow this distance

    if(moved){
        totalDistance +=distance; //updates distance travelled
    }

    return moved;
}
```



First Assignment

- Based on this example sand will be posted later today.
- It will be due next Friday.





OLLSCOIL NA GAILLIMHE
UNIVERSITY OF GALWAY

CT2106

Object Oriented Programming



Dr. Frank Glavin
Room 404, IT Building
Frank.Glavin@UniversityofGalway.ie
School of Computer Science

University
ofGalway.ie

Variables and Types

- A variable is a symbol used to store a value
 - E.g. $x = 5$
- In **strongly typed** language, you have to tell the compiler/interpreter what **type** the variable is
- The Compiler/Interpreter knows how much space to allocate it in memory



Java Primitive Variables

Type	Size	Range
boolean	1 bit	true or false
byte	8 bits	[-128, 127]
short	16 bits	[-32,768, 32,767]
char	16 bits	['\u0000', '\uffff'] or [0, 65535]
int	32 bits	[-2,147,483,648 to 2,147,483,647]
long	64 bits	$[-2^{63}, 2^{63}-1]$
float	32 bits	32-bit IEEE 754 floating-point
double	64 bits	64-bit IEEE 754 floating-point



Default values

- Each primitive variable has a **default value**.
- The default value is used **only when the variable is used as a field (instance variable)**
- If the field is not explicitly assigned a value, the default value is used
- For example, the default value for an **int** variable is 0 (zero)

Useful example and summary:

<https://www.codejava.net/java-core/the-java-language/java-default-initialization-of-instance-variables-and-initialization-blocks>



Example

```
public class Bicycle
{
    // instance variables - replace the example below with your own
    private int speed;

    /**
     * Constructor for objects of class Bicycle
     */
    public Bicycle()
    {
        // note how the speed variable is not initialised
        // it will us the default value for an int, zero
    }

    /**
     * @return value of speed field
     */
    public int getSpeed()
    {
        return speed;
    }
}
```



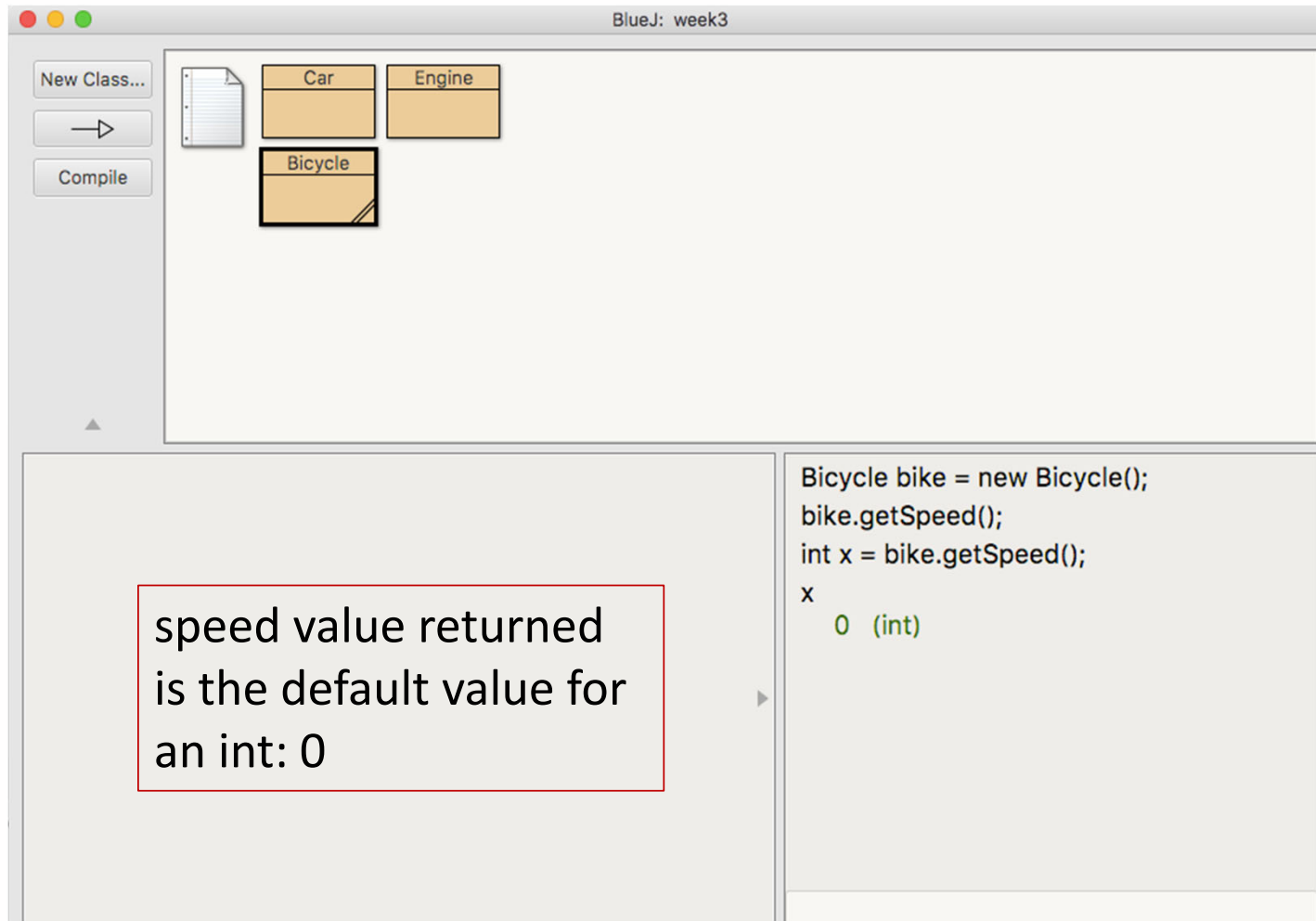
BlueJ: week3

New Class...
→
Compile

Car Engine
Bicycle

```
Bicycle bike = new Bicycle();  
bike.getSpeed();  
int x = bike.getSpeed();  
x  
0 (int)
```

speed value returned
is the default value for
an int: 0

The image shows a screenshot of the BlueJ IDE interface. At the top, the window title is "BlueJ: week3". On the left side, there are three buttons: "New Class...", a right-pointing arrow, and "Compile". The main workspace is divided into two sections. The upper section contains a class diagram with three class boxes: "Car", "Engine", and "Bicycle". The "Bicycle" box is highlighted with a black border. The lower section is split into two panes. The left pane contains a text box with a red border that says "speed value returned is the default value for an int: 0". The right pane shows a code snippet: "Bicycle bike = new Bicycle();", "bike.getSpeed();", "int x = bike.getSpeed();", followed by a variable declaration "x" and its value "0 (int)".

Default Values

- The Code pad in Blue J automatically initialises **variables just as if they were instance variables**.
- This will not happen in a true Java program!
- But it is useful for learning the default values.



Default Values

Your turn – type a variable of each type into

Code Pad

E.g type: **int y;**

Hit return

then type: **y**

Hit return

Write down the default
value returned for each
type

Type	Size
boolean	1 bit
byte	8 bits
short	16 bits
char	16 bits
int	32 bits
long	64 bits
float	32 bits
double	64 bits



Starting Example

```
int y;  
Note: Codepad variables are automatically initialized  
in the same way as instance fields.  
y  
0 (int)
```



```
int y;
```

Note: Codepad variables are automatically initialized in the same way as instance fields.

```
y
```

```
0 (int)
```

```
boolean bool;
```

```
bool
```

```
false (boolean)
```

```
byte b;
```

```
b
```

```
0 (byte)
```

```
short s;
```

```
s
```

```
0 (short)
```

```
char c;
```

```
c
```

```
'\u0000' (char)
```

```
long lg;
```

```
lg
```

```
0 (long)
```

```
float f;
```

```
f
```

```
0.0 (float)
```

```
double d;
```

```
d
```

```
0.0 (double)
```



Java Primitive Variables

Default values

Type	Size	Range	Default
boolean	1 bit	true or false	false
byte	8 bits	[-128, 127]	0
short	16 bits	[-32,768, 32,767]	0
char	16 bits	['\u0000', '\uffff'] or [0, 65535]	'\u0000'
int	32 bits	[-2,147,483,648 to 2,147,483,647]	0
long	64 bits	$[-2^{63}, 2^{63}-1]$	0
float	32 bits	32-bit IEEE 754 floating-point	0.0
double	64 bits	64-bit IEEE 754 floating-point	0.0



Reference/Object Types

- A reference type is a data type that's based on a class rather than on one of the primitive types that are built into the Java language.
- In fact, there are four categories of reference type:
 - Object Types
 - Interface Types
 - Enum Types
 - Array Types
- For now, we will focus on Object types, the others will follow easily



Object Reference Type: Key points

- A variable that is a reference type is a variable that **points to an object**
- A primitive variable contains **the value** of the primitive type .
- e.g. **int x = 7;** x contains the int value 7
- A reference variable **contains the value of the memory location** of an object
- E.g. **Wheel wheel = new Wheel();**
- The **wheel** variable contains the value of the memory location of the new Wheel object



Key point to Remember

- A reference variable **does not** contain the value of the object
- A reference variable contains the value of the memory location of the object
- It is a **pointer**



Null Default value

- The default value of all reference variables is **null**;
- null is a special value in Java
- It means 'No object'
- When you first declare a reference variable, its value is null

```
Bicycle bike; // declaring a reference variable called bike of type Bicycle
bike // what's the value of bike?
    null
Bicycle bike2; // declaring another reference variable of type Bicycle
bike2 // what's the value of bike2
    null
```



NullPointerException

- One of the most common errors generated when running a program in Java is **NullPointerException**
- This error is thrown when your program encounters a reference variable that has not been initialised
- This means that the variable points to its default value = **null**
- Your program then tries to get the object that the variable is pointing to to do something.
- But the object doesn't exist. Variable actually points to null.
- This causes Java to generate a **NullPointerException**



Example

Using your previously defined Bicycle class, type the following into Code Pad

```
Bicycle bike1; //bike1 points to null  
Bicycle bike2; // bike1 points to null;
```

```
bike1 = new Bicycle();  
bike2 = new Bicycle();
```

bike1, bike2 are assigned to point to the Bicycle objects just initialised

```
bike1 = null;  
bike2 = null;
```

bike1, bike2 again point to null

What has happened to the previously initialised Bicycle objects?



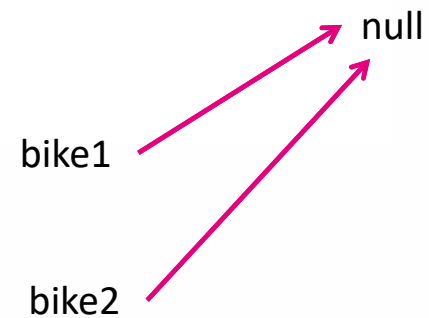
Understanding References

```
Bicycle bike1; //bike1 points to null  
Bicycle bike2; // bike1 points to null;
```

```
bike1 = new Bicycle();  
bike2 = new Bicycle();
```

```
bike1 = bike2;
```

```
bike1 = null;  
bike2 = null;
```



Understanding References

```
Bicycle bike1; //bike1 points to null  
Bicycle bike2; // bike1 points to null;
```

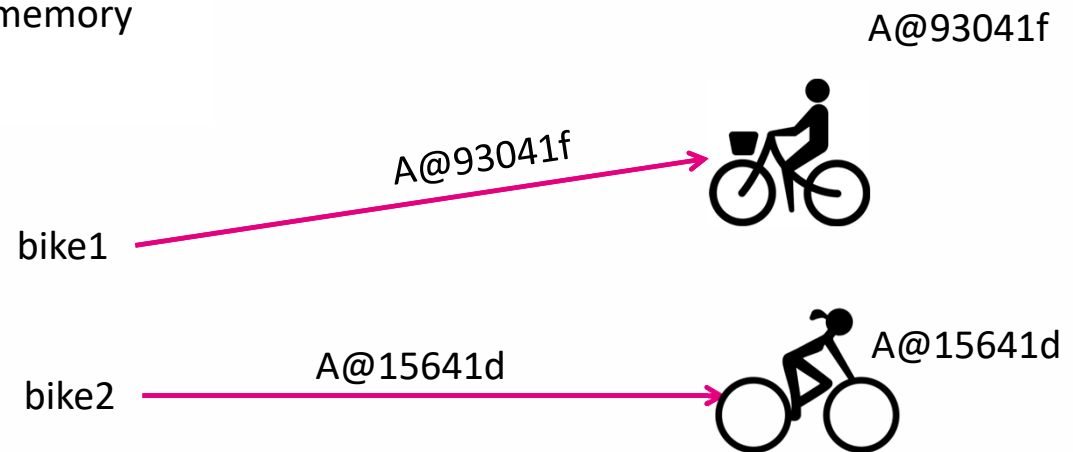
```
bike1 = new Bicycle();  
bike2 = new Bicycle();
```

```
bike1 = bike2;
```

The actual values of bike1,
bike2 are memory
locations

```
bike1 = null;  
bike2 = null;
```

Objects stored at
memory locations



Understanding References

```
Bicycle bike1; //bike1 points to null  
Bicycle bike2; // bike1 points to null;
```

```
bike1 = new Bicycle();  
bike2 = new Bicycle();
```

```
bike1 = bike2;
```

bike1 now takes the
same value as bike2

```
bike1 = null;  
bike2 = null;
```

Objects stored at
memory locations



A@93041f

bike1

A@15641d

bike2

A@15641d



A@15641d



Understanding References

```
Bicycle bike1; //bike1 points to null  
Bicycle bike2; // bike1 points to null;
```

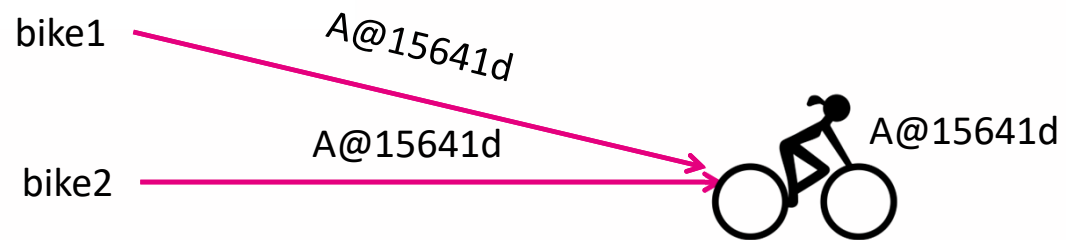
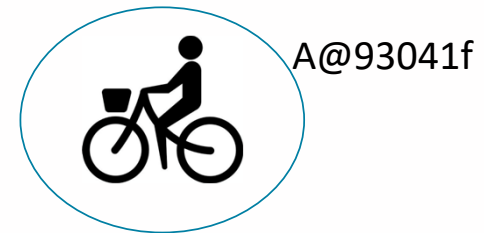
```
bike1 = new Bicycle();  
bike2 = new Bicycle();
```

```
bike1 = bike2;
```

bike1 now takes the
same value as bike2

```
bike1 = null;  
bike2 = null;
```

What happens to
this object?



Understanding References

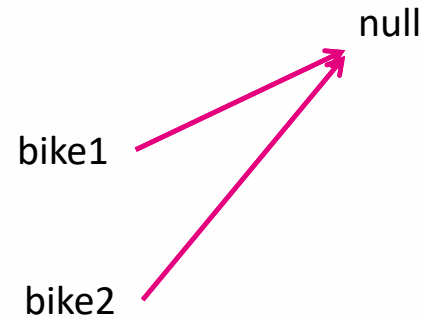
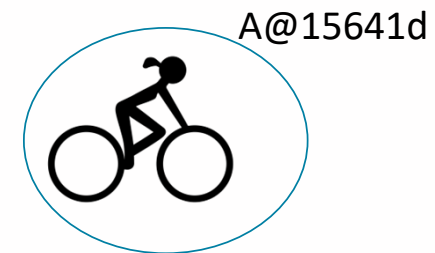
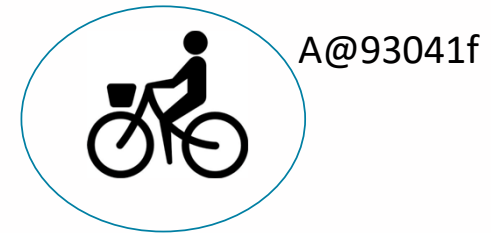
```
Bicycle bike1; //bike1 points to null  
Bicycle bike2; // bike1 points to null;
```

```
bike1 = new Bicycle();  
bike2 = new Bicycle();
```

```
bike1 = bike2;
```

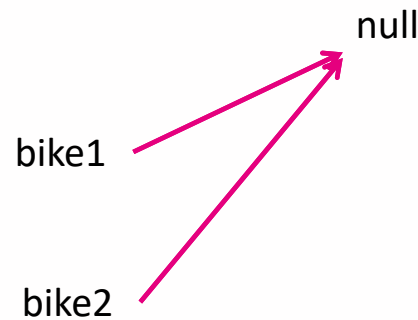
```
bike1 = null;  
bike2 = null;
```

What happens to
both these object?

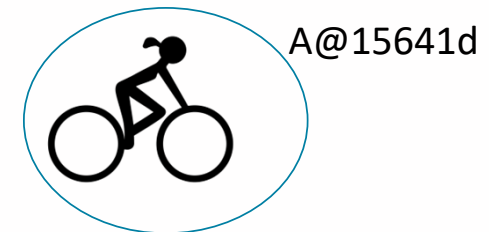
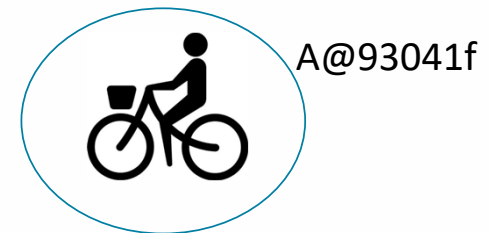


Memory Leak

This is what is called a memory leak.
In this case, you have two objects occupying memory and you have not deallocated them from memory
In fact, there is no way to deallocate them in Java!
So how do you deal with lost objects?



What happens to both these object?



Garbage Collector

- The Garbage collector is part of the JRE's memory management system
- It runs in the background keeping track of the live objects in a program and marking the rest as garbage
- The data in these marked areas are subsequently deleted, freeing up memory



Understanding References

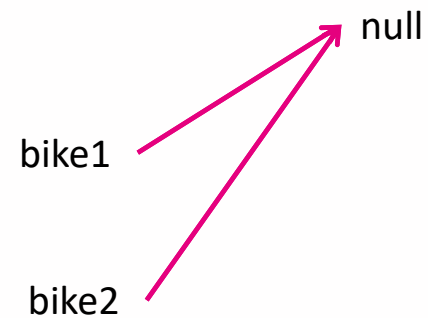
```
Bicycle bike1; //bike1 points to null  
Bicycle bike2; // bike1 points to null;
```

```
bike1 = new Bicycle();  
bike2 = new Bicycle();
```

```
bike1 = bike2;
```

```
bike1 = null;  
bike2 = null;
```

Garbage Collector



Understanding References

```
Bicycle bike1; //bike1 points to null  
Bicycle bike2; // bike1 points to null;
```

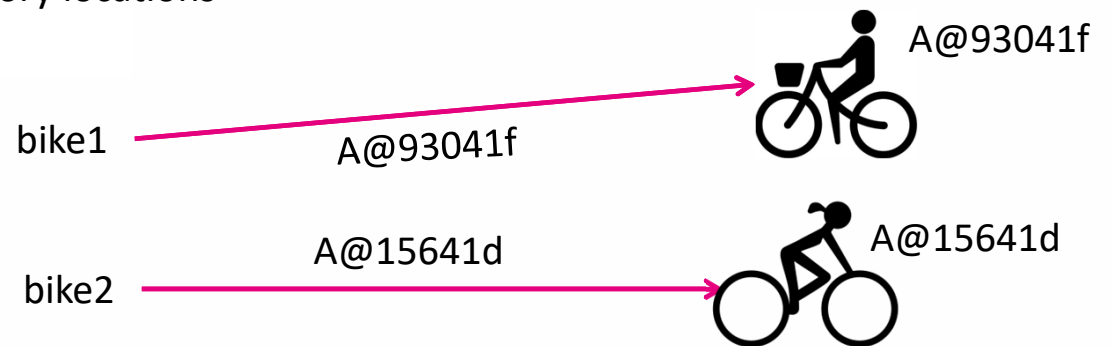
```
bike1 = new Bicycle();  
bike2 = new Bicycle();
```

```
bike1 = bike2; The actual values of bike1,  
bike2 are memory locations  
bike1 = null;  
bike2 = null;
```

Garbage collector



A@93041f live
A@15641d live



Understanding References

```
Bicycle bike1; //bike1 points to null  
Bicycle bike2; // bike1 points to null;
```

```
bike1 = new Bicycle();  
bike2 = new Bicycle();
```

```
bike1 = bike2;
```

Bike1 now takes the same value as bike2

```
bike1 = null;  
bike2 = null;
```

Garbage collector

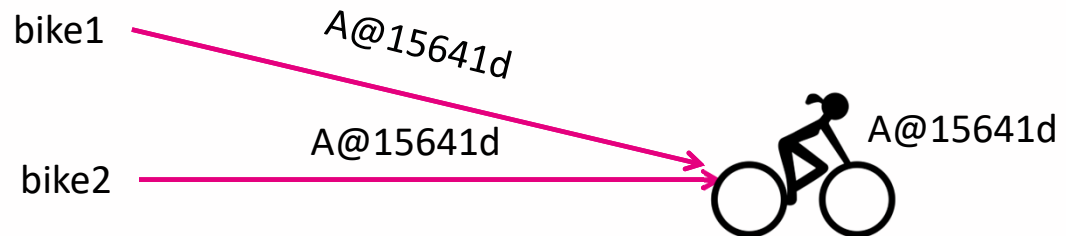


A@93041f **unreferenced**

A@15641d **live**



A@93041f



Understanding References

```
Bicycle bike1; //bike1 points to null  
Bicycle bike2; // bike1 points to null;
```

```
bike1 = new Bicycle();  
bike2 = new Bicycle();
```

```
bike1 = bike2;
```

```
bike1 = null;  
bike2 = null;
```

Bike1 now takes the same value as bike2

Garbage collector

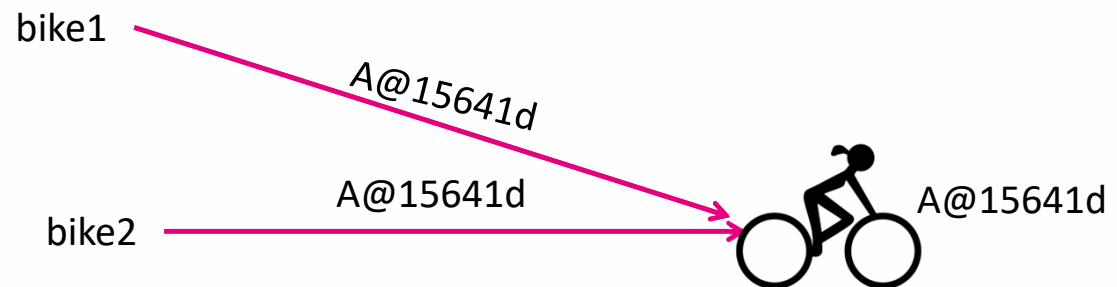
A@93041f **delete**

A@15641d **live**

A@93041f



Yum, yum



OLLSCOIL NA GAILLIMH
UNIVERSITY OF GALWAY

Understanding References

```
Bicycle bike1; //bike1 points to null  
Bicycle bike2; // bike1 points to null;
```

```
bike1 = new Bicycle();  
bike2 = new Bicycle();
```

```
bike1 = bike2;
```

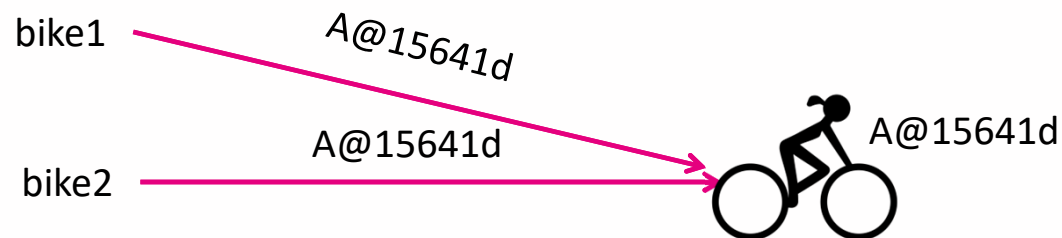
Bike1 now takes the same value as bike2

```
bike1 = null;  
bike2 = null;
```

Garbage collector



A@15641d live



OLLSCOIL NA GAILLIMHIE
UNIVERSITY OF GALWAY

Understanding References

```
Bicycle bike1; //bike1 points to null  
Bicycle bike2; // bike1 points to null;
```

```
bike1 = new Bicycle();  
bike2 = new Bicycle();
```

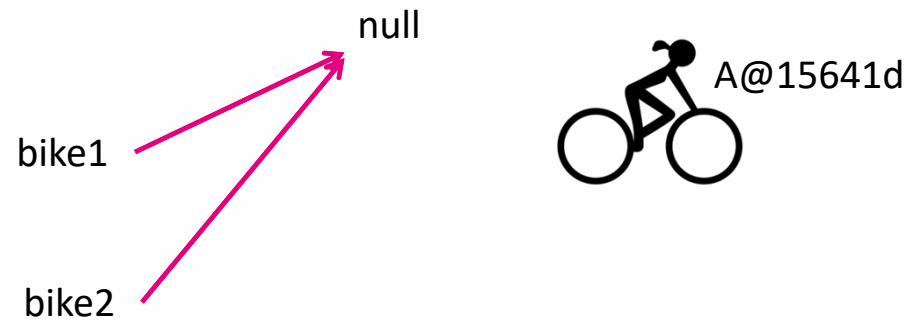
```
bike1 = bike2;
```

```
bike1 = null;  
bike2 = null;
```

Garbage collector



A@15641d **unreferenced**



Understanding References

```
Bicycle bike1; //bike1 points to null  
Bicycle bike2; // bike1 points to null;
```

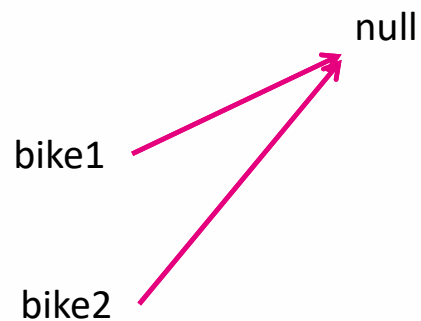
```
bike1 = new Bicycle();  
bike2 = new Bicycle();
```

```
bike1 = bike2;
```

```
bike1 = null;  
bike2 = null;
```

Garbage collector

A@15641d delete



Understanding References

```
Bicycle bike1; //bike1 points to null  
Bicycle bike2; // bike1 points to null;
```

```
bike1 = new Bicycle();  
bike2 = new Bicycle();
```

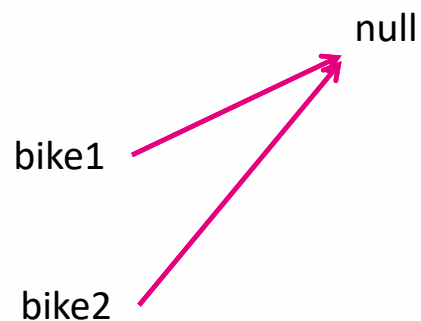
```
bike1 = bike2;
```

```
bike1 = null;  
bike2 = null;
```

Garbage collector



waiting...for its next
unreferenced object



OLLSCOIL NA GAILLIMHÉ
UNIVERSITY OF GALWAY

True or False?

The value of a variable in Java can be

- 1) A primitive
- 2) A reference value
- 3) An object

```
int x = 2;  
Bicycle bike = new Bicycle(1,2,3);
```



False

The value of a variable  Java can be

1) A primitive 

2) A reference value

3) An object 

```
int x = 2;  
Bicycle bike = new Bicycle(1,2,3);
```

The value of a variable is **never** an object. However, it can take a reference value to an object



Assignment Steps

```
Car car = new Car("X7");  
Engine engine = new Engine("DR9", 43);  
car.add(engine);  
Wheel wheel = new Wheel ("Wichelin15", 15);  
car.add(wheel);  
car.setFuel(100);  
car.run();  
car.getDistance();
```



Test-driven development

The code before is our **test**

It specifies the minimum we have to do to demonstrate the overall program works as per the problem specification

Once the code we have written outputs what we want, we can stop

This will be version 1 of our assignment



What we know

We have three classes: **Car, Engine and Wheel**

We know the **properties** of each class

We have composition relationships between them

- Car composed of Engine

- Engine composed of Wheel

We know that they have to create a few methods in each class so that objects can call each other in order for the program to deliver the functionality we require



Approach

Test-driven development = **incremental approach** to solving a problem

Incrementally create Stub classes and Stub methods so that your code compiles and runs at all times

To start with, it may run – but it may do nothing interesting.

Gradually we add functionality – making sure it compiles and runs

We keep doing this until we achieve our minimum criteria for success

In this case - we want to print out the distance achieved





OLLSCOIL NA GAILLIMHE
UNIVERSITY OF GALWAY

CT2106

Object Oriented Programming



Dr. Frank Glavin
Room 404, IT Building
Frank.Glavin@UniversityofGalway.ie
School of Computer Science

University
ofGalway.ie

Ideas Encountered So Far

- An object is responsible for how its data is represented internally.
- Constructors are special methods used to bootstrap an object into existence – and generally used to *initialise* its state.
- Java has two types of variables
 - Primitive types
 - Reference types
- The Java Garbage Collector runs in the background monitoring which objects are live (referenced). The remainder of objects in memory are marked for deletion



OOP modelling

- A major part of OOP is modelling the problem. The goal is to identify:
 - The principle objects in the problem domain
 - We model these as a classes
- The responsibility of each of these objects
 - What does it do?
- What are the collaborations between objects?
 - What other object does it communicate with?



When attempting an OOP solution

- Identify the main (real) concepts in the problem domain
- Our objective is to produce a simplified class diagram
 - **classes** represent real-world entities
 - **associations** represent collaborations between the entities
 - **attributes** represent the data held about entities
 - **generalization** can be used to simplify the structure of the model (we'll look at this later)



Perspective

- This should be a fairly quick process
- You can expect your model to be incomplete on your first iteration
- There may well be important conceptual objects in the domain that you do not discover until implementation



Identify the Objects/Classes

- **Write down a description of what your program is required to do?**
- Identify and list the **nouns** in each description
- The goal is to identify
 - Potential Objects
 - Attributes of objects
- **Some** of these objects may eventually be modelled as software classes and objects
- This is the beginning of a process of identification, refinement and (re-)modelling



Example: Stage 1: Identify nouns

A Java program for handling a customer online transaction

The customer verifies the items in their shopping cart. Customer provides payment and address to process the sale. The System validates the payment and responds by confirming the order, and provides the order number that the customer can use to check on the order status. The System will send the customer a copy of the order details by email

- Nouns = candidate objects



Identify nouns

A Java program for handling a customer online transaction

The customer verifies the *items* in their shopping cart. Customer provides payment and address to process the *sale*. The System validates the payment and responds by confirming the order, and provides the order number that the customer can use to check on the order status. The System will send the customer a copy of the order details by email



- Nouns = candidate objects

Customer	Order
Item	Order Number
Shopping Cart	Order Status
Payment	Order Details
Address	Email
Sale	System

- Identify duplicates (e.g sale and order)
- You may find yourself combining/splitting some of these concepts
- Which are properties?



Customer
Item
Shopping Cart
Payment
Address
~~Sale~~

Order
~~Order Number~~
~~Order Status~~
~~Order Details~~
Email
~~System~~

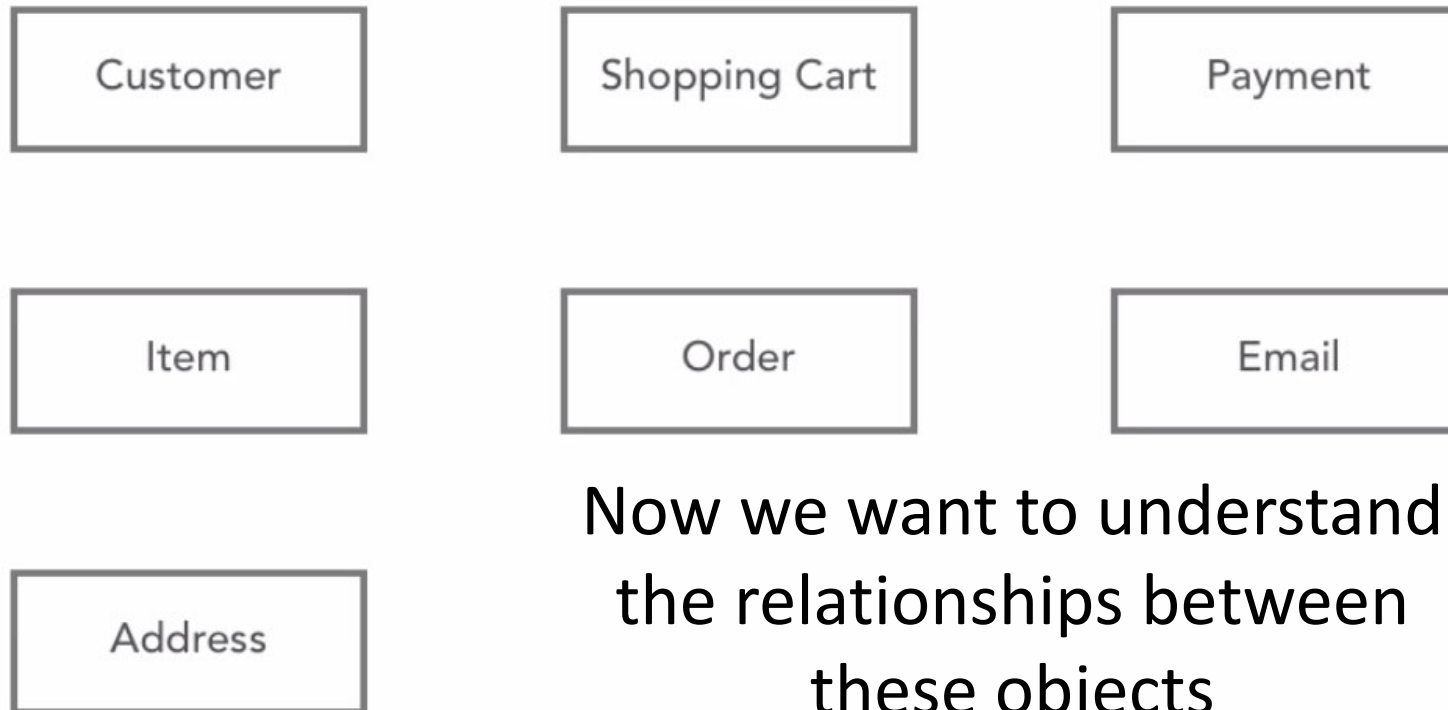
Attributes of Order



Avoid global objects such as System
These will tend to accumulate too much responsibility



A simple **class diagram** of the conceptual objects



Now we want to understand the relationships between these objects



Stage 2: Identify associations

Initially associations may be identified by the relationships in the description

A Java program for handling a customer online transaction

The customer verifies the items in their shopping cart. Customer provides payment and address to process the sale. The System validates the payment and responds by confirming the order, and provides the order number that the customer can use to check on the order status. The System will send the customer a copy of the order details by email

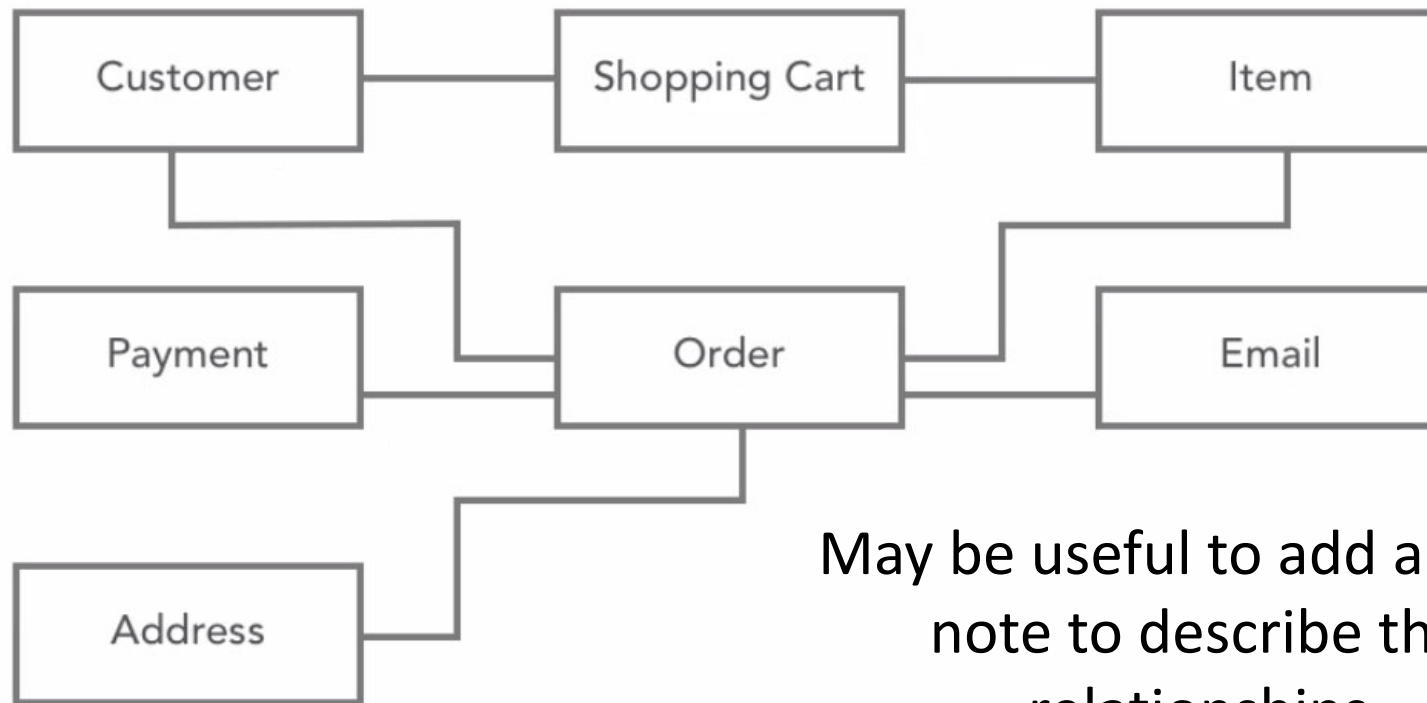


Potential Associations

Customer, Shopping Cart
Shopping Cart, Item
Customer, Order
Order, Payment, Address, Email



OLLSCOIL NA GAILLIMHÉ
UNIVERSITY OF GALWAY



May be useful to add a short note to describe the relationships



Stage 3: Identify Responsibilities

Examine the **verbs** and **verb phrases** in each Use Case

A Java program for handling a customer online transaction

The customer verifies the items in their shopping cart. Customer provides payment and address to process the sale. The System validates the payment and responds by confirming the order, and provides the order number that the customer can use to check on the order status. The System will send the customer a copy of the order details by email



Stage 3: Identify Responsibilities

Examine the **verbs** and **verb phrases** in each Use Case

A Java program for handling a customer online transaction

The customer **verifies the items** in their shopping cart. Customer **provides payment and address** to **process the sale**. The System **validates the payment** and responds by **confirming the order**, and provides the order number that the customer can use to **check on the order status**. The System will **send** the customer a copy of the order details **by email**



Stage 3: Identify Responsibilities

- Examine the **verbs** and **verb phrases** in each Use Case
 - Verify Items
 - Provide Payment and address
 - Process sale
 - Validate Payment
 - Confirm order
 - Provide order number
 - Check order status
 - Send order details by email

However, it may not be obvious from the description **where** these responsibilities should reside



Stage 4: Assign Responsibilities

Determine which responsibilities belong to which class

Candidate responsibilities

Verify Items

Provide Payment and address

Process sale

Validate Payment

Confirm order

Provide order number

Check order status

Send order details by email

Candidate Classes

Customer

Shopping Cart

Payment

Order

Email

Address



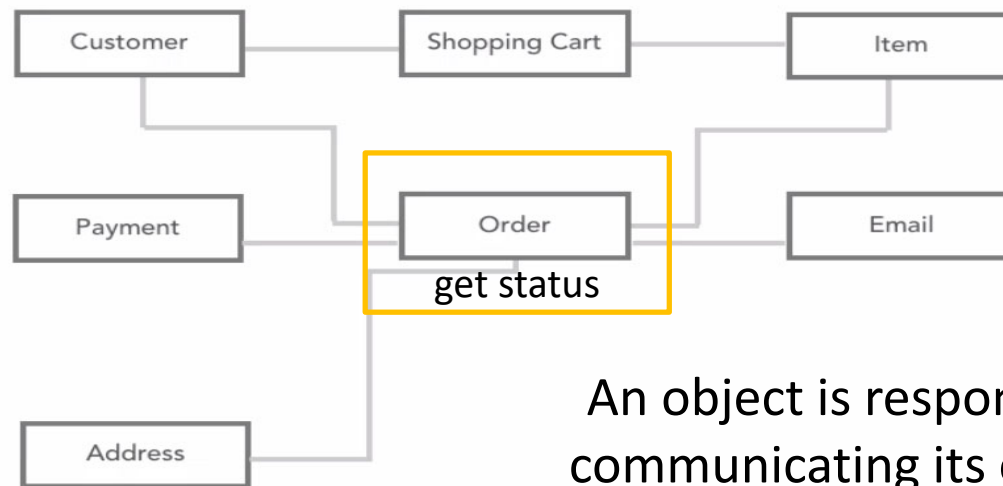
OO Principles

Consider the following principles when assigning responsibilities

1. **An Object is responsible for its own data**
An object has responsibility for communicating its state
2. **Single Responsibility Principle: Each Class should have a single responsibility**
All its services should be aligned with that responsibility



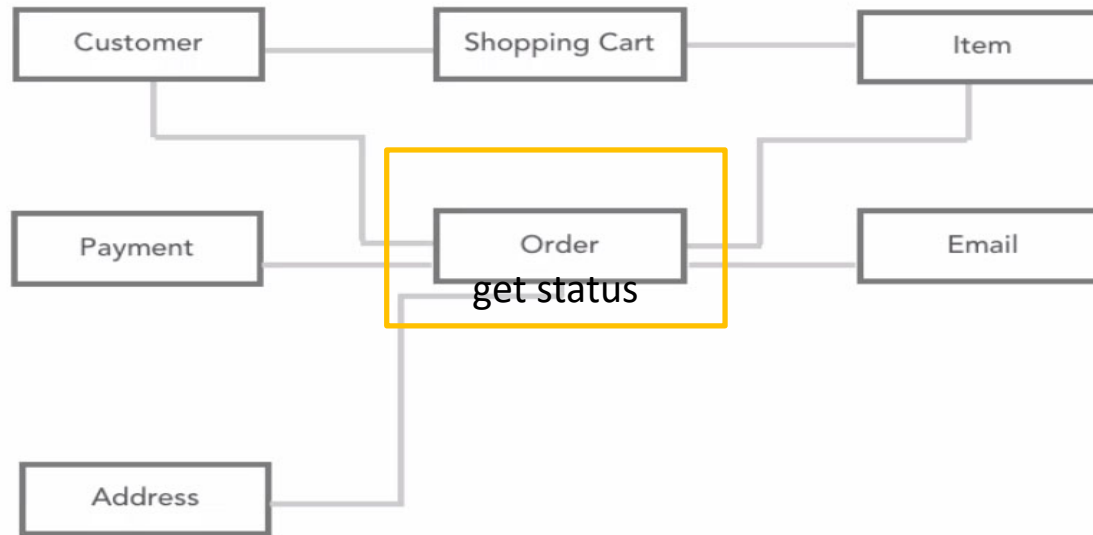
- Consider the responsibility **Check order status**
- The real customer initiates this action
- However which object should be responsible for checking the order status?



An object is responsible for communicating its own state



Now Attach method to the classes



- Verify Items
- Provide Payment and address
- Process sale
- Validate Payment
- Confirm order
- Provide order number
- ~~Check order status~~
- Send order details by email



Recall OO Principles

1. An Object is responsible for its own data

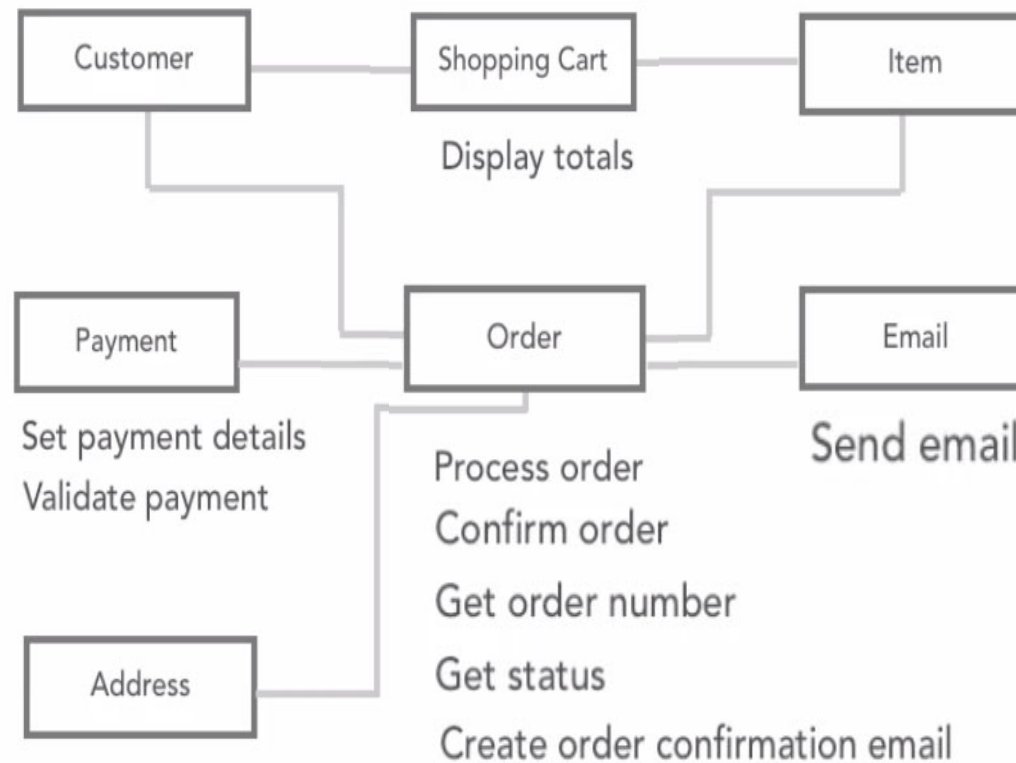
An object has responsibility for communicating its state

2. Single Responsibility Principle: Each Class should have a **single** responsibility

All its services should be aligned with that responsibility



Assigning Responsibilities



- Verify items
- Provide payment and address
- Process sale
- Validate payment
- Confirm order
- Provide order number
- Check order status
- Send order details email



Perspective

Some objects seems to have no/few responsibilities – not a problem

The scenario we presented focused on one aspect of the overall

The diagram doesn't show which entities initiate actions

A common mistake in OO modelling is to assign too much responsibility to the actor (the user)

Another common mistake is to assign lots of responsibility to a centralised System object



Working with 'System'

A Java program for handling a customer online transaction

The customer verifies the items in their shopping cart. Customer provides payment and address to process the sale. The **System validates the payment** and responds by confirming the order, and provides the order number that the customer can use to check on the order status. The **System will send the customer a copy of the order details by email**



Working with 'System'

On first inspection it may seem that you need a centralised System object with many responsibilities.

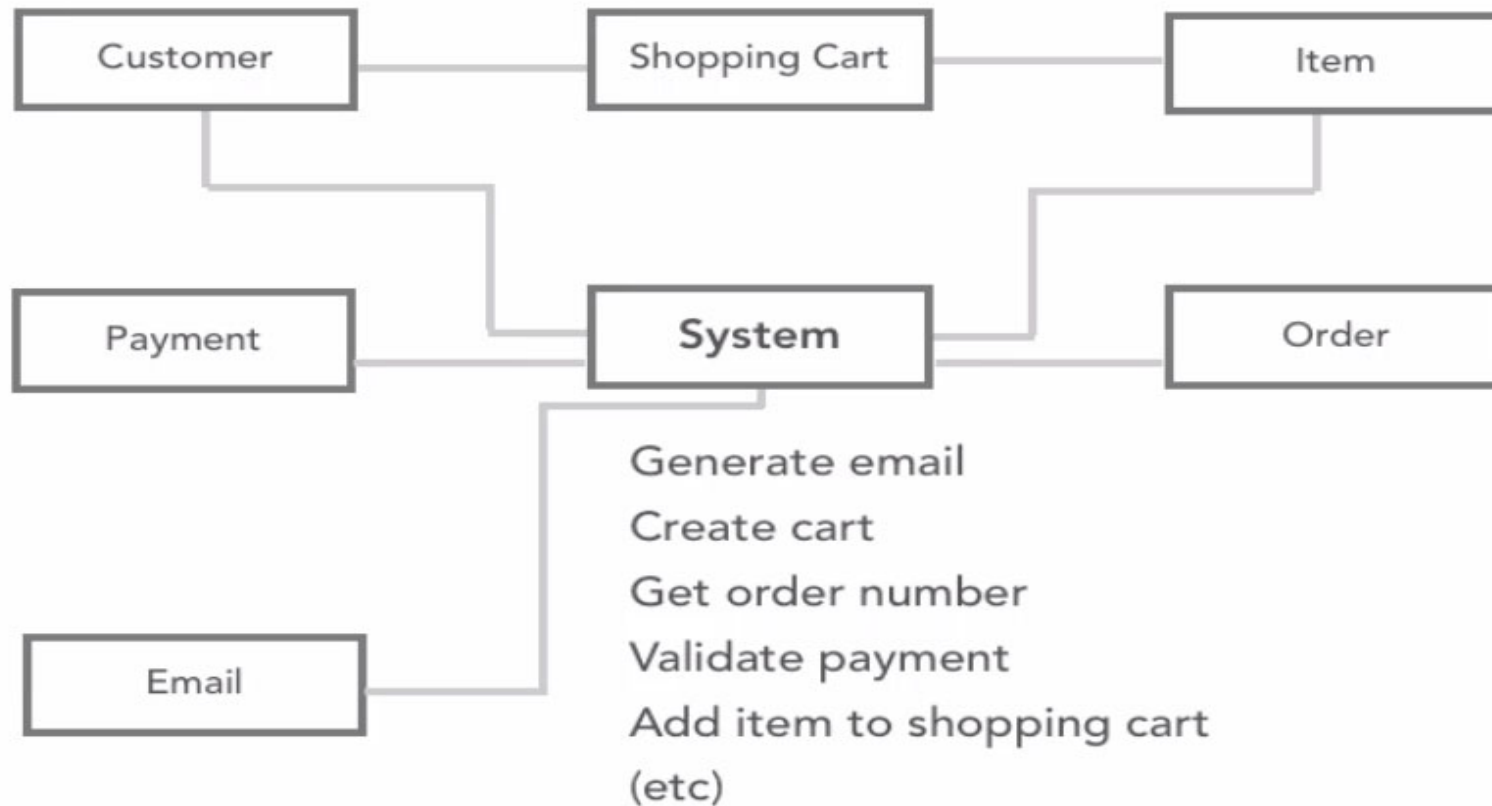
Often this will be a poor design decision

“System validates payment” = “some part of the system validates payment”

Your job is to figure out which part of the System should have this responsibility



Avoid 'God Objects': Objects that know and do too much



https://en.wikipedia.org/wiki/God_object

God object

From Wikipedia, the free encyclopedia

For an object worshiped as a god, see [Idol](#).



This article includes a [list of references](#), related reading or [external links](#), **but its sources remain unclear because it lacks [inline citations](#)**. Please help to [improve](#) this article by [introducing](#) more precise citations. *(March 2012)* ([Learn how and when to remove this template message](#))

In [object-oriented programming](#), a **god object** is an [object](#) that *knows too much* or *does too much*. The god object is an example of an [anti-pattern](#).

A common programming technique is to [separate](#) a large problem into several smaller problems (a [divide and conquer strategy](#)) and create solutions for each of them. Once the smaller problems are solved, the big problem as a whole has been solved. Therefore a given object for a small problem need only know about itself. Likewise, there is only one set of problems an object needs to solve: its *own* problems.

In contrast, a program that employs a god object does not follow this approach. Most of such a program's overall functionality is coded into a single "all-knowing" object, which maintains most of the information about the entire program, and also provides most of the [methods](#) for manipulating this data. Because this object holds so much data and requires so many methods, its role in the program becomes god-like (all-knowing and all-encompassing). Instead of program objects communicating among themselves directly, the other objects within the program rely on the single god object for most of their information and interaction. Since this object is tightly [coupled](#) to (referenced by) so much of the other code, maintenance becomes more difficult than it would be in a more evenly divided programming design. Changes made to the object for the benefit of one routine can have unintended effects on other unrelated routines.

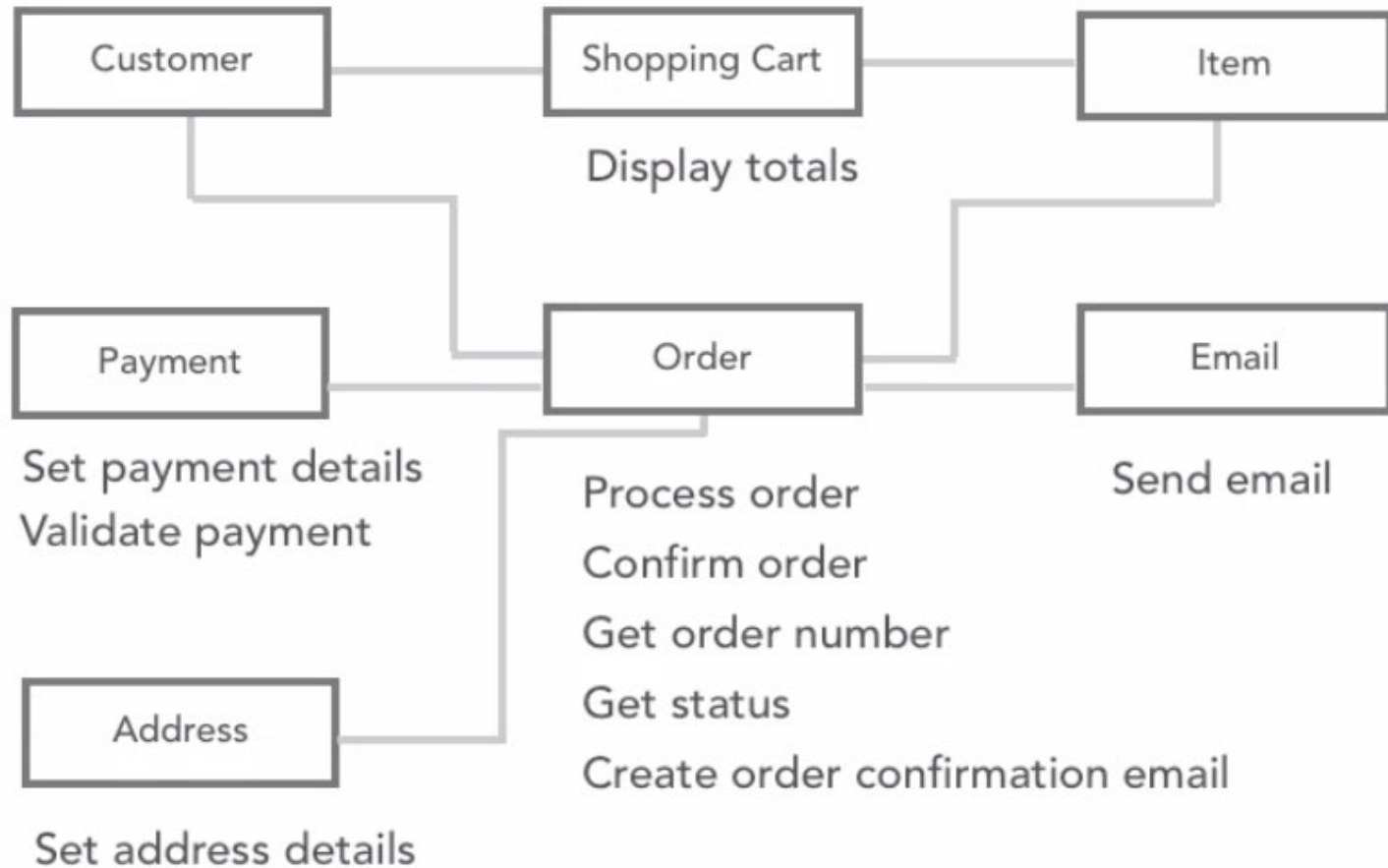
A god object is the object-oriented analogue of failing to use [subroutines](#) in [procedural programming languages](#), or of using far too many [global variables](#) to store state information.

Whereas creating a god object is typically considered bad programming practice, this technique is occasionally used for tight programming environments (such as [microcontrollers](#)), where the performance increase and centralization of control are more important than maintainability and programming elegance.



OLLSCOIL NA GAILLIMHIE
UNIVERSITY OF GALWAY

Responsibilities should be distributed



Lecture Summary

- A major part of OOP is modelling the problem
- Identifying the principle **objects**, their **responsibilities** and **collaborations** between objects
- Key idea is to develop a description of how the program ought to work
 - Extract nouns -> candidate classes/objects
 - Examine relationships in text - > object associations
 - Examine verbs -> possible methods
 - Assign responsibilities to classes
- Consider the single responsibility principle, and object encapsulation (in charge of its own state)
- Avoid God objects





OLLSCOIL NA GAILLIMHE
UNIVERSITY OF GALWAY

CT2106

Object Oriented Programming



Dr. Frank Glavin
Room 404, IT Building
Frank.Glavin@UniversityofGalway.ie
School of Computer Science

University
ofGalway.ie

OOP Modelling

- A major part of OOP is modelling the problem
- The goal is to identify:
- The principle **objects** in the problem domain
 - We model these as a **classes**
- The responsibility of each these objects
 - What does it do?
- What are the collaborations between objects
 - What other object does it communicate with



When attempting an OOP solution

- Identify the main (real) concepts in the problem domain
- Our objective is to produce a simplified class diagram
 - **classes** represent real-world entities
 - **associations** represent collaborations between the entities
 - **attributes** represent the data held about entities
 - **generalization** can be used to simplify the structure of the model (we'll look at this later)



Identify the objects/Classes

- Write down a description of what your program is required to do
- Identify and list the nouns in each description
- The goal is to identify
 - **Potential Objects**
 - **Attributes of objects**
- Some of these objects may eventually be modelled as software classes and objects
- This is the beginning of a process of identification, refinement and (re-)modelling



Program Description

A Java program for handling a customer online transaction

The customer verifies the items in their shopping cart. Customer provides payment and address to process the sale. The System validates the payment and responds by confirming the order, and provides the order number that the customer can use to check on the order status. The System will send the customer a copy of the order details by email

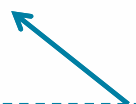


Customer
Item
Shopping Cart
Payment
Address
~~Sale~~

Sale = Order

Order
~~Order Number~~
~~Order Status~~
~~Order Details~~
Email
~~System~~

Attributes of Order



Avoid global objects such as System
These will tend to accumulate too much responsibility



A simple class diagram of the conceptual objects



Now we want to understand
the relationships between
these objects



Stage 2: Identify Associations

Initially, associations may be identified by the relationships in the description

A Java program for handling a customer online transaction

The customer verifies the items in their shopping cart. Customer provides payment and address to process the sale. The System validates the payment and responds by confirming the order, and provides the order number that the customer can use to check on the order status. The System will send the customer a copy of the order details by email



Potential Associations

Customer, Shopping Cart

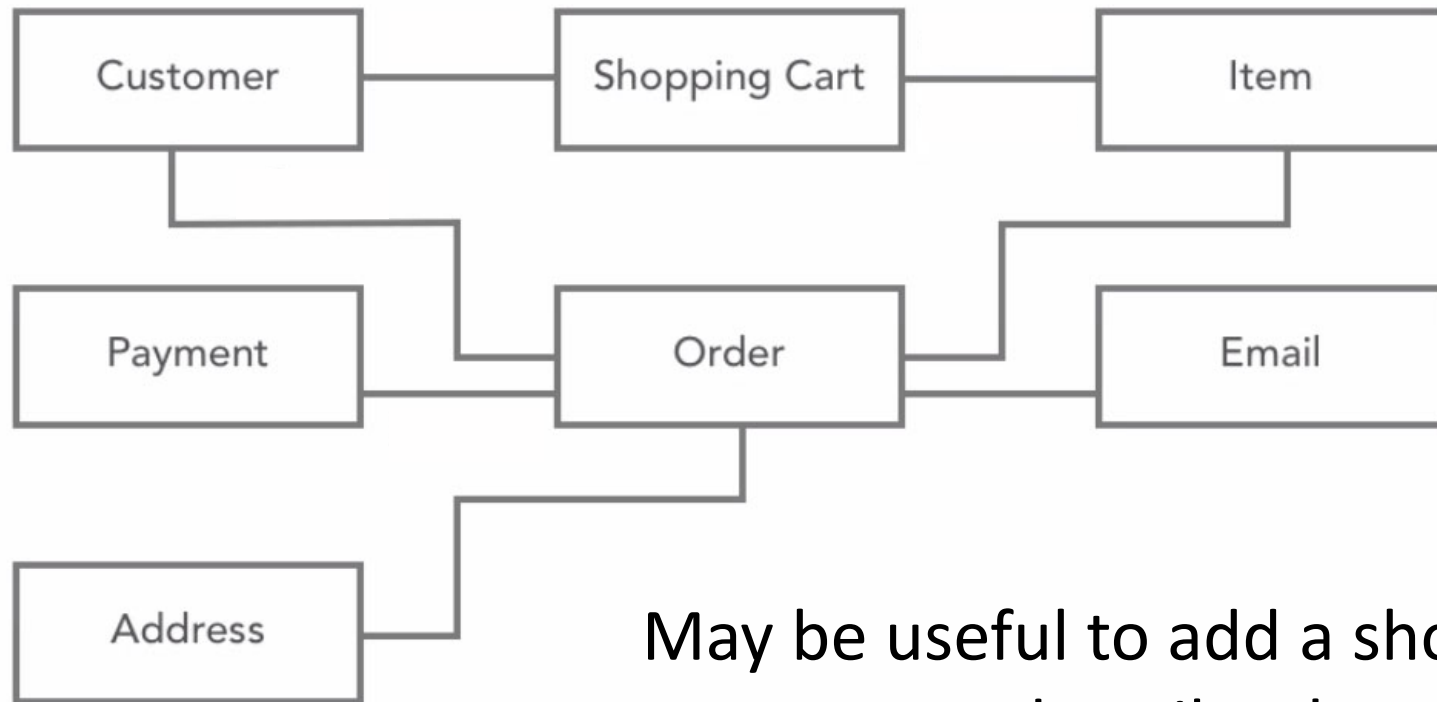
Shopping Cart, Item

Customer, Order

Order, Payment, Address, Email



OLLSCOIL NA GAILLIMHÉ
UNIVERSITY OF GALWAY



May be useful to add a short note to describe the relationships



Stage 3: Identify Responsibilities

Examine the **verbs** and **verb phrases** in each Use Case

Verify Items

Provide Payment and address

Process sale

Validate Payment

Confirm order

Provide order number

Check order status

Send order details by email

However, it may not be obvious from the description **where** these responsibilities should reside



Stage 4: Assign Responsibilities

Determine which responsibilities belong to which class

Candidate responsibilities

Verify Items
Provide Payment and address
Process sale
Validate Payment
Confirm order
Provide order number
Check order status
Send order details by email

Candidate Classes

Customer
Shopping Cart
Payment
Order
Email
Address



Recall OO Principles

1. **An Object is responsible for its own data**

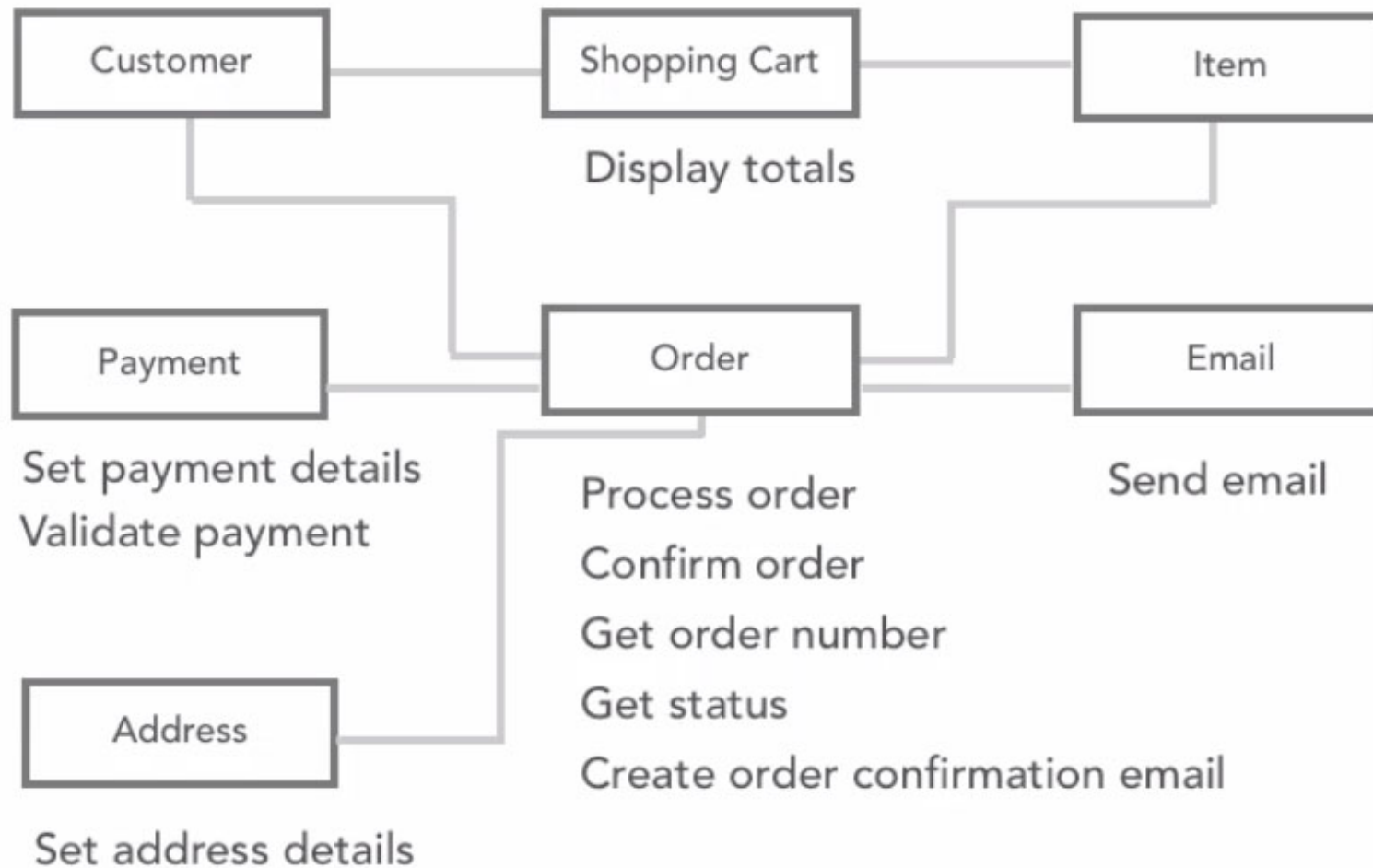
- An object has responsibility for communicating its state

2. **Single Responsibility Principle:** Each Class should have a **single** responsibility

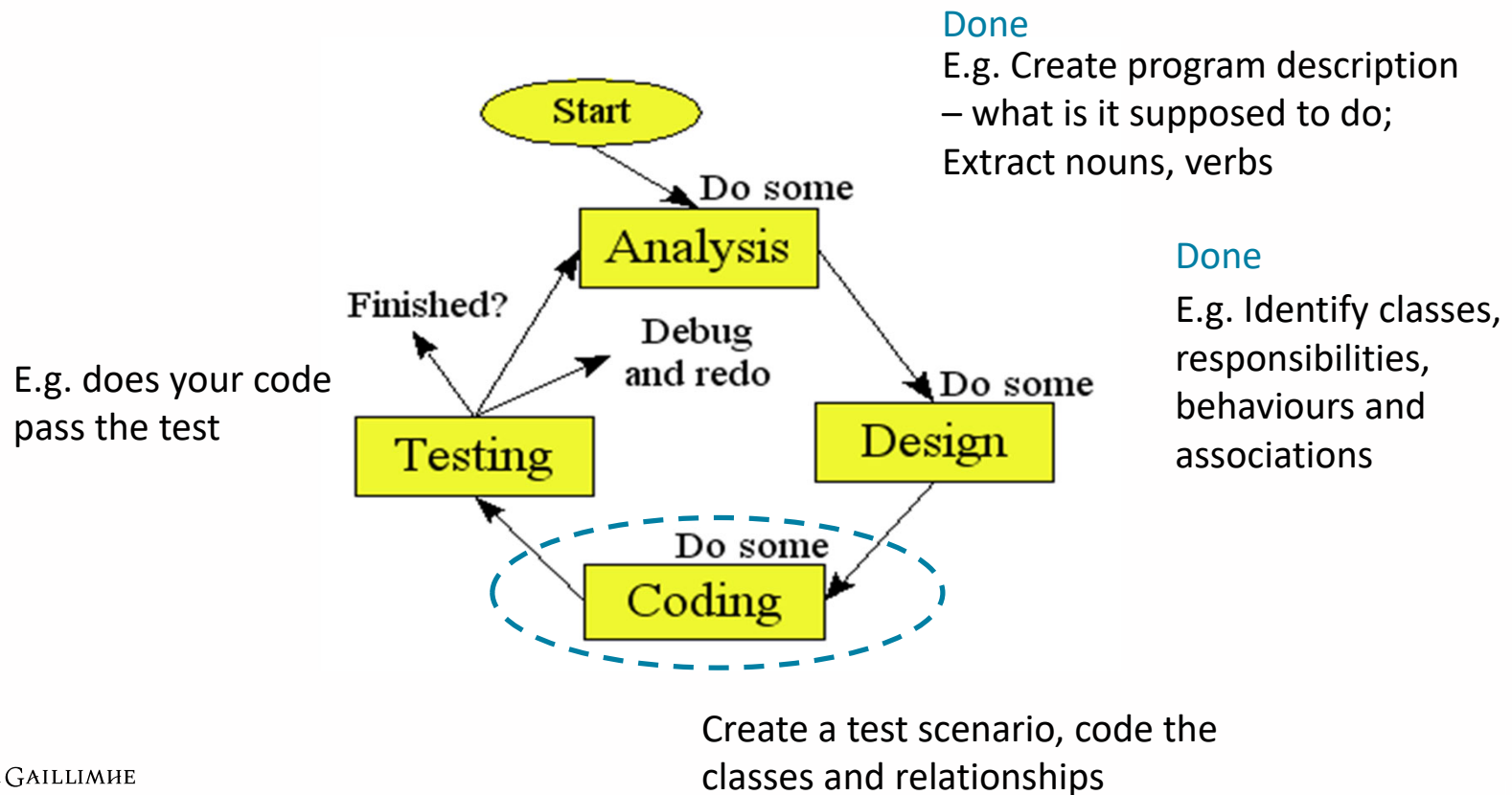
- All its services should be aligned with that responsibility



Responsibilities should be distributed



Iterative, Incremental Development



Starting to Code: Set yourself an objective

Firstly create a test class, to test how the candidate classes **should** work together

You should set a **measurable objective** for your test class to achieve
i.e. If your classes work correctly they should calculate/output a particular number or message

In fact, you did this for Assignment 1



Test Scenario Code

```
Car car = new Car("X7");  
Engine engine = new Engine("DR9", 43);  
car.add(engine);  
Wheel wheel = new Wheel ("Wichelin15", 15);  
car.add(wheel);  
car.setFuel(100);  
car.drive();  
car.getDistance();
```

Test Output

This program should output how far a particular Car configuration can travel given a full tank of fuel (say 100 units)

Assumption

If the Test code can output the correct distance value for the fuel value, then the code works



Test Code Scenario v1

1. Create Customer object
 2. Create Shopping Cart object for the Customer
 3. Add 3 items with known cost to cart
 4. Finalise the cart and create an order
 5. Add a delivery address for the order
 6. Add a payment type
 7. Validate the payment
 8. If successful, email the customer with a success email and the cost of the purchased items
- Our code passes the test scenario if an email is created with a message giving the correct total;**



Turning this into code

1. Write a basic test class to test the scenario. The class will have a main method
2. Line by line, write the outline code of the scenario
3. As you write it, you should try to compile it.
4. In each step, do enough to make it compile

At the end of this process you will have a rough outline of v1 of the overall solution.

It may not run properly – but you will have made many of the key modelling/implementation decisions

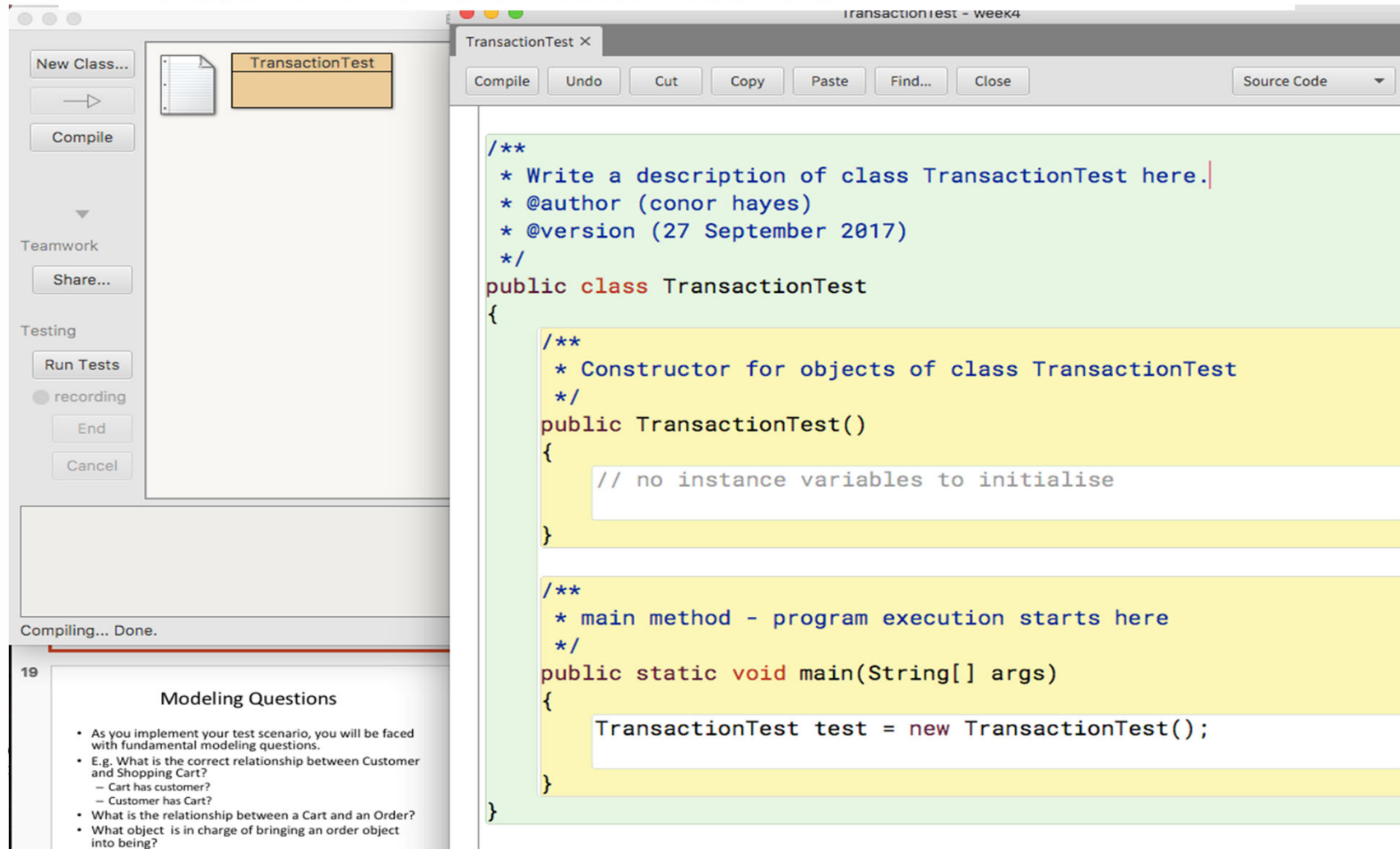


Modeling Questions

- As you implement your test scenario, you will be faced with fundamental modeling/implementation questions.
- E.g. What is the correct relationship between Customer and Shopping Cart?
 - Cart has a customer?
 - Customer has a Cart?
- What is the relationship between a Cart and an Order?
- How does an order object get access to the shopping cart data?
- How do you prevent new items being added to a Cart, once an order (based on the cart) has been initialised



1. Write a basic test class to test the scenario The class will have a main method



The screenshot shows an IDE window titled "TransactionTest - week4". The main editor displays the following Java code:

```
/**
 * Write a description of class TransactionTest here.
 * @author (conor hayes)
 * @version (27 September 2017)
 */
public class TransactionTest
{
    /**
     * Constructor for objects of class TransactionTest
     */
    public TransactionTest()
    {
        // no instance variables to initialise
    }

    /**
     * main method - program execution starts here
     */
    public static void main(String[] args)
    {
        TransactionTest test = new TransactionTest();
    }
}
```

The IDE interface includes a "New Class..." button, a "Compile" button, and a "Teamwork" section with a "Share..." button. The "Testing" section has a "Run Tests" button and "recording" status with "End" and "Cancel" buttons. A status bar at the bottom indicates "Compiling... Done." and a slide titled "19 Modeling Questions" is visible in the background.



1. Write a basic test class to test the scenario The class will have a main method

- Create a **method** to hold the code for each scenario
- Alternatively, You could write the code directly into the main method
- However, having a separate method for each scenario allows you to test multiple scenarios at once



```
/**
 * main method - program execution starts here
 */
public static void main(String[] args)
{
    TransactionTest test = new TransactionTest();
    test.transaction1(); // each method can contain a different transaction scenario
    test.transaction2();
    test.transcation3();
}
```

- To get started, get transaction1 working
- Create stub code for each of these methods in order to have your code compile
- For now, we'll only work on transaction1



```

/**
 * main method - program execution starts here
 */
public static void main(String[] args)
{
    TransactionTest test = new TransactionTest();
    test.transaction1(); // each method can contain a different transaction scenario
    test.transaction2();
    test.transaction3();
}

public void transaction1(){
    // the body of our first code scenario will go in here
    //This will be the code that tests if our order transaction classes work
}

public void transaction2(){
    // we can put the body of another code scenario here
    // for now we'll just focus on putting code into transaction1
}

public void transaction3(){
    // we can put the body of yet another code scenario here
    // for now we'll just focus on putting code into transaction1
}

```



```
public void transaction1(){
```

```
    //the body of our first code scenario will go in here
```

```
    //This will be the code that tests if our order transaction classes work
```

Goal: turn the steps below into code within the transaction1 method

1. Create Customer object
 2. Create Shopping Cart object for the Customer
 3. Add 3 items with known cost to cart
 4. Finalize the cart and create an order
 5. Add a delivery address for the order
 6. Add a payment type
 7. Validate the payment
 8. If successful, email the customer with a success email and the cost of the purchased items
- Our code passes the test scenario if an email is created with a message giving the correct total;**



Method: proceed in steps

1. Add a line of code
 2. Do the minimum required to get it to compile
 3. Do 1 and 2 until finished the scenario
- At this point you will have compiling stub code for all the classes you need.
 - Your code will still require work to make it run correctly – but you have at least 50% of the work done.
 - For every change you make, make sure to recompile your code



Create a Customer object

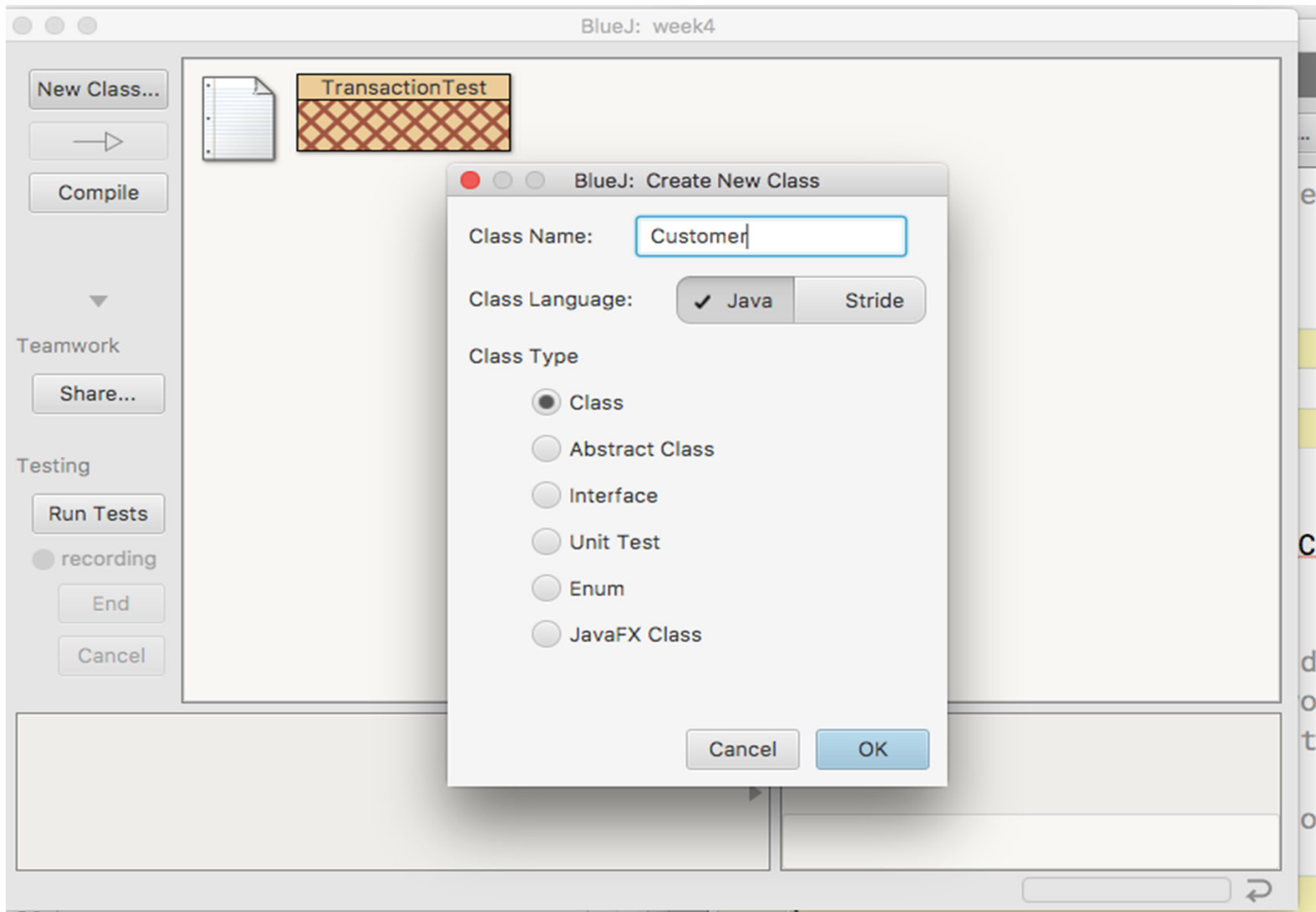
Just write a line of code to create a Customer object

```
public void transaction1(){  
  
    Customer customer = new Customer();  
  
    //When you write this code BlueJ will complain that it can't find a customer class  
    // Therefore your code won't compile  
    // Use this as the prompt to create a simple Customer class  
    // Compile the code  
    // Now consider, what properties should a customer object have  
  
}
```

cannot find symbol - class Customer

Your program won't compile because there is no Customer class - **yet**






```
public class Customer
{
    // instance variables or 'fields' go here
    // What fields should a customer object have?
    // It depends really on what the role is of the customer object

    /**
     * Constructor for objects of class Customer
     */
    public Customer()
    {
        // initialise the instance variables - but what are they?
    }
}
```



A Customer class

1. Question you should ask yourself: **What are the properties and responsibilities of the Customer object in this programme.**
2. List the properties that a Customer might have
3. These will be the fields of the Customer class
4. Create the field variables - what type will each of these have?



Shopping Cart class

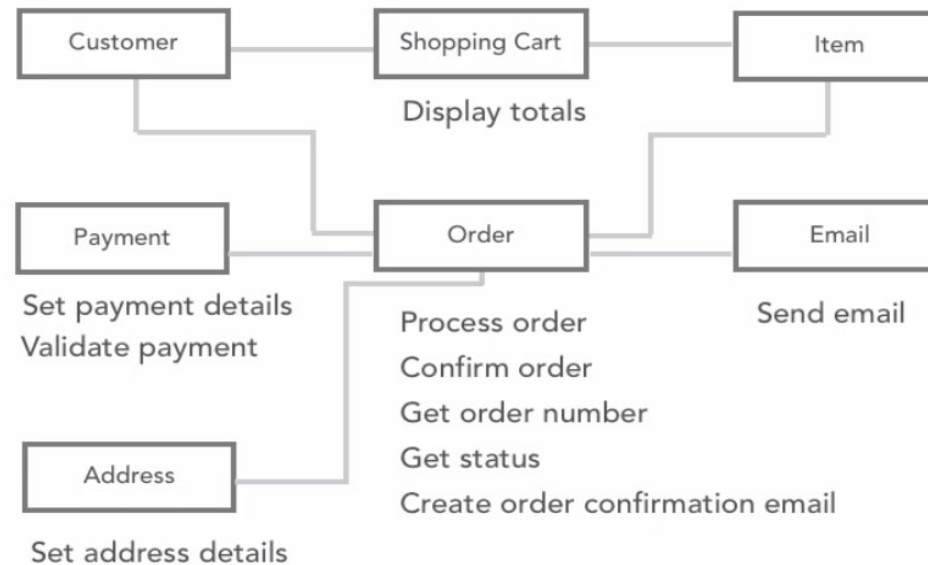
Step 2 of the scenario:

“Create Shopping Cart object for the Customer”



ShoppingCart

- What is the role of the shopping Cart?
- What are its properties/responsibilities/relationships etc
- Recall our earlier analysis



Shopping Cart and Customer

- What is the relationship between ShoppingCart and Customer
 - a) Does a Customer have a Cart?
 - b) Does a Cart have a Customer ?
- Justify the decision you will make



Shopping Cart Requirements

- add Items
 - remove items
 - print out the the Items in it
 - display totals
 - **lock it** so that items cannot be added/removed from it
 - We want to be able to clear it completely.
-
- **Write the Shopping Cart code**





OLLSCOIL NA GAILLIMHE
UNIVERSITY OF GALWAY

CT2106

Object Oriented Programming



Dr. Frank Glavin
Room 404, IT Building
Frank.Glavin@UniversityofGalway.ie
School of Computer Science

University
ofGalway.ie

Yesterday's lecture

- Create a test class to test your code
- Line by line create the stub code and methods
- Until you have the outline of your programme compiling
- Even getting to this stage will force you to make many of the key decisions for your solution
 - Object properties and methods
 - Object collaboration



Revision (1)

- **Class**

- A **blueprint** or **template** or **set of instructions** to build a specific type of **object**.
- Every object is built from a class.
- Each class should be designed and programmed to realise a **single** responsibility
-

- **Method**

- A method is the equivalent of a function.
- Methods are the actions that perform operations on a variable (Fields)



Revision (2)

Encapsulation

- Binding 'object' state (fields) and behaviour (methods) together.
- Creating a class means you are doing encapsulation.
- The core idea is to:
 - Hide the implementation details from users
 - No method outside the class can access it directly.
- How?
 - **Private**
 - Protected

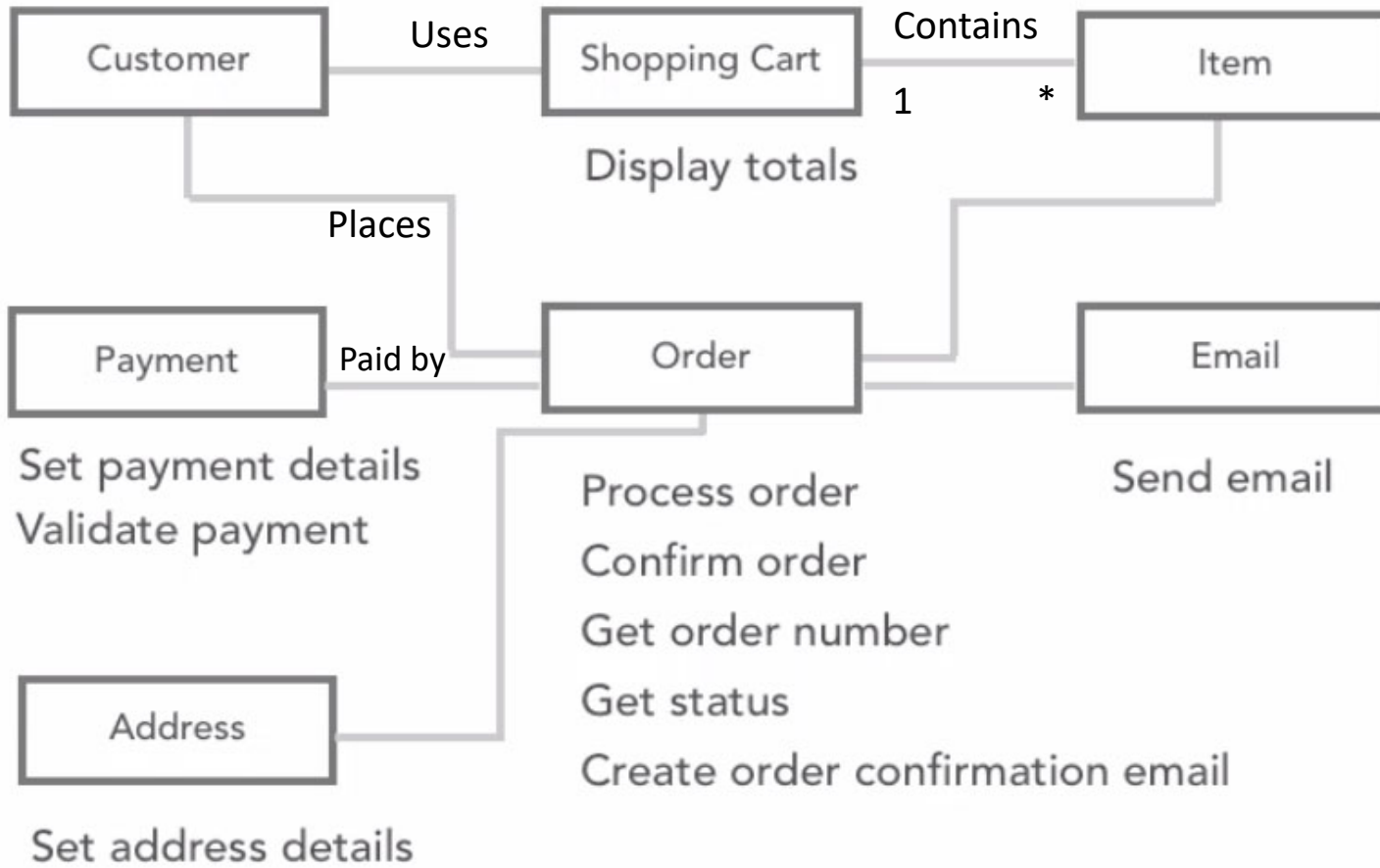


Program Description

A Java program for handling a customer online transaction

The customer verifies the items in their shopping cart. Customer provides payment and address to process the sale. The System validates the payment and responds by confirming the order, and provides the order number that the customer can use to check on the order status. The System will send the customer a copy of the order details by email





Test Code Scenario v1

1. Create Customer object
2. Create Shopping Cart object for the Customer
3. Add 3 items with known cost to cart
4. Finalise the cart and create an order
5. Add a delivery address for the order
6. Add a payment type
7. Validate the payment
8. If successful, email the customer with a success email and the cost of the purchased items

Our code passes the test scenario if an email is created with a message giving the correct total;



We created a test class

```
public class TransactionTest
{
    /**
     * main method to execute the TransactionTest methods
     */
    public static void main(String[] args)
    {
        TransactionTest test = new TransactionTest();
        test.transaction1(); // calls the method with our test scenario
    }

    public void transaction1(){

        // write your test code here

    }
}
```



```
public void transaction1(){  
    //the body of our first code scenario will go in here  
    //This will be the code that tests if our order transaction classes work
```

Goal: turn the steps below into code (within the transaction1 method)

1. Create Customer object
 2. Create Shopping Cart object for the Customer
 3. Add 3 items with known cost to cart
 4. Finalise the cart and create an order
 5. Add a delivery address for the order
 6. Add a payment type
 7. Validate the payment
 8. If successful, email the customer with a success email and the cost of the purchased items
- Our code passes the test scenario if an email is created with a message giving the correct total;**



```
}
```

Method: Proceed in steps

1. Add a line of code
2. Do the minimum required to get it to compile
3. Do 1 and 2 until finished the full scenario



Test Code Scenario v1

1. Create Customer object
2. Create Shopping Cart object for the Customer
3. Add 3 items with known cost to cart
4. Finalise the cart and create an order
5. Add a delivery address for the order
6. Add a payment type
7. Validate the payment
8. If successful, email the customer with a success email and the cost of the purchased items



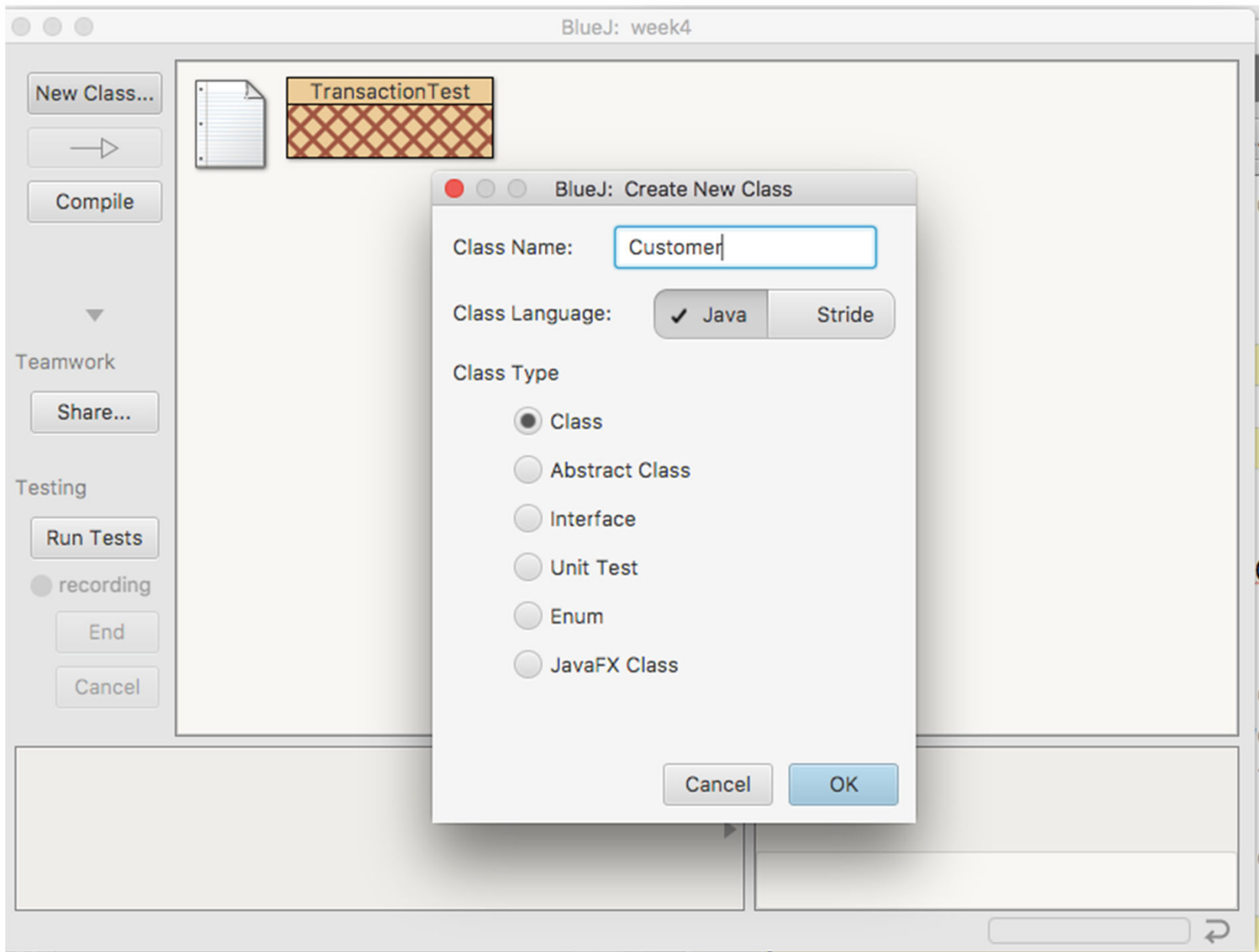
Create a Customer object

Just write a line of code to create a Customer object

```
public void transaction1(){  
  
    Customer customer = new Customer();  
  
    //When you write this code BlueJ will complain that it can't find a customer class  
    // Therefore your code won't compile  
    // Use this as the prompt to create a simple Customer class  
    // Compile the code  
    // Now consider, what properties should a customer object have  
  
}
```

Your program won't compile because there is no Customer class - **yet**





```
public class Customer
{
    // instance variables or 'fields' go here
    // What fields should a customer object have?
    // It depends really on what the role is of the customer object

    /**
     * Constructor for objects of class Customer
     */
    public Customer()
    {
        // initialise the instance variables - but what are they?
    }
}
```



Customer

What are the properties and responsibilities of the Customer object in this programme?

The Customer object holds the data about the Customer data
Any object can request information about the Customer from it



```
public class Customer {
    private String firstName;
    private String surName;
    private String emailAddress;
    private final long customerId;

    public Customer(String firstName, String surName, String emailAddress){
        this.firstName = firstName;
        this.surName = surName;
        this.emailAddress = emailAddress;
        customerId = makeCustomerId();
    }

    public long getId() {
        return customerId;
    }

    public String getFirstName() {
        return firstName;
    }

    public String getSurName() {
        return surName;
    }
}
```



Update your code in the TransactionTest class

```
public void transaction1(){  
  
    //1. Create New Customer  
    Customer customer = new Customer("Niamh", "O'Leary", "niamhol@zmail.com");  
  
    // 2. Create a Shopping Cart for the Customer
```



Test Code Scenario v1

- ~~1. Create Customer object~~
- 2. Create Shopping Cart object for the Customer**
3. Add 3 items with known cost to cart
4. Finalise the cart and create an order
5. Add a delivery address for the order
6. Add a payment type
7. Validate the payment
8. If successful, email the customer with a success email and the cost of the purchased items



ShoppingCart Class

- Now add the code for the Shopping Cart

```
public void transaction1(){  
  
    //1. Create New Customer  
    Customer customer = new Customer("Niamh", "O'Leary", "niamhol@zmail.com");  
  
    // 2. Create a Shopping Cart for the Customer  
    ShoppingCart cart = new ShoppingCart(customer);  
}
```

- Your code won't compile, because you haven't yet created a Shopping cart class
- **This is your cue to create the ShoppingCart class**



Shopping Cart fields?

- *What fields might a Shopping Cart have? Briefly explain the reason for each field.*
 - **cartId**: a unique numerical Id for the Cart
 - **time**: the date/time it was created
 - **items**: to hold the items in the cart
 - **total** : to hold the total for the items in the cart



Shopping Cart behaviours?

- *Methods belonging to a shopping cart?*
- Here are some potential ones:
 - add Item
 - remove item
 - print out the the Items in it
 - display total
 - **lock it** so that items cannot be added/removed from it
 - clear the cart.



Customer / Cart Relationship?

- a) *Does a Customer have a Cart?*
- b) *Does a Cart have a Customer ?*



Class exercise: Create a Shopping Cart class

Fields:

cartId: numerical

time: String

items: holds a collection

total: numerical

customer: ref type Customer

The Item class is in the next slide – you can download it from Blackboard

Methods:

addItem

removeItem

getTotal

getCartId

getCustomer

printItems

close

clear



Item Class

```
public class Item {  
    private String name;  
    private int price;  
    private long itemId;  
  
    public Item(String itemName, long id) {  
        name = itemName;  
        itemId = id;  
    }  
  
    public void setPrice(int price){  
        this.price = price;  
    }  
  
    public int getPrice() {  
        return price;  
    }  
  
    @Override  
    public String toString(){  
        String out = "Item Id: " + itemId + "\t" + name + "\tPrice: " + price;  
        return out;  
    }  
}
```



addItem

After you have defined the fields start with defining the *addItem* method
See the tutorial on Collections for help with this
adding an object (in this case, an Item) to a collection



Assignment 2

- Based on the code we've written so far
- Remember:
 - Code in increments
 - Always set your code a measureable objective
 - Such as the test scenario mentioned earlier
 - Create a version 0.1 with basic functionality – this will teach you a lot about the problem



Lecture Wrap-up (1)

- Much of OOP is about making modeling decisions
- A model is a simplified representation of reality
- Core modeling decisions: what are the objects, what data do they contain, what are their responsibilities, what are their associations with each other



Lecture Wrap-up (2)

- Start by identifying the objects and relationships in the problem domain – these are candidate objects for your code solution
- It is important to set your code an objective or test before writing the code
- Create the stub code for your classes/methods
- Compile and develop step by step





OLLSCOIL NA GAILLIMHE
UNIVERSITY OF GALWAY

CT2106

Object Oriented Programming



Dr. Frank Glavin
Room 404, IT Building
Frank.Glavin@UniversityofGalway.ie
School of Computer Science

University
ofGalway.ie

Last Week

- Much of OOP is about making good modeling decisions
- A model is a simplified representation of reality
- Core modeling decisions: what are the objects, what are their responsibilities, what are their associations with each other
- Start by identifying the objects and relationships in the problem domain – these are candidate objects
- It is important to set your code an objective or test before writing the code
- Create the stub code for your classes
- Development, particularly OO development is **incremental** and **iterative**



This lecture

This lecture will prepare the groundwork for the next major topic we cover in OOP:

- **Inheritance**

Today's topics:

- Object equivalence



- Open BlueJ
- Create a new Project
- Make sure Code Pad is displayed
- (View-> Show Code Pad)



Instructions 1

1. Create a String variable **str1** to hold a String value “Java”
2. Type **str1** into CodePad. It should return the value “Java”
3. Create another String variable **str2** to hold a String value “Ja”
4. Create another String variable **str3** to hold a String value “va”
5. Create another String object **str4** to hold the String value when **str3** is added to **str2**
6. Type **str4** into CodePad. It should return the value “Java”



Instructions 2

You are now going to check for the equality of the values of **str1** and **str4**

1. Write an **if** statement to test if **str1** has the same value as **str4**
2. The if statement should print out **true** if **str1** has the same value as **str4** and **false** if they do not print out the same value

(Hold down the Shift and Enter keys to enter more than one line in CodePad)



Hint

```
int x = 8;
int y = 9;

if (x==y) {
    System.out.println("true");
} else{
    System.out.println("false");
}
```



How many wrote something like this?

```
String str1 = "Java";  
String str2 = "Ja";  
String str3 = "va";  
String str4 = str2+str3;  
  
if(str1==str4){  
    System.out.println("true");  
} else{  
    System.out.println("false");  
}
```



What will the output be?

```
if(str1==str4){  
    System.out.println("true");  
} else{  
    System.out.println("false");  
}
```



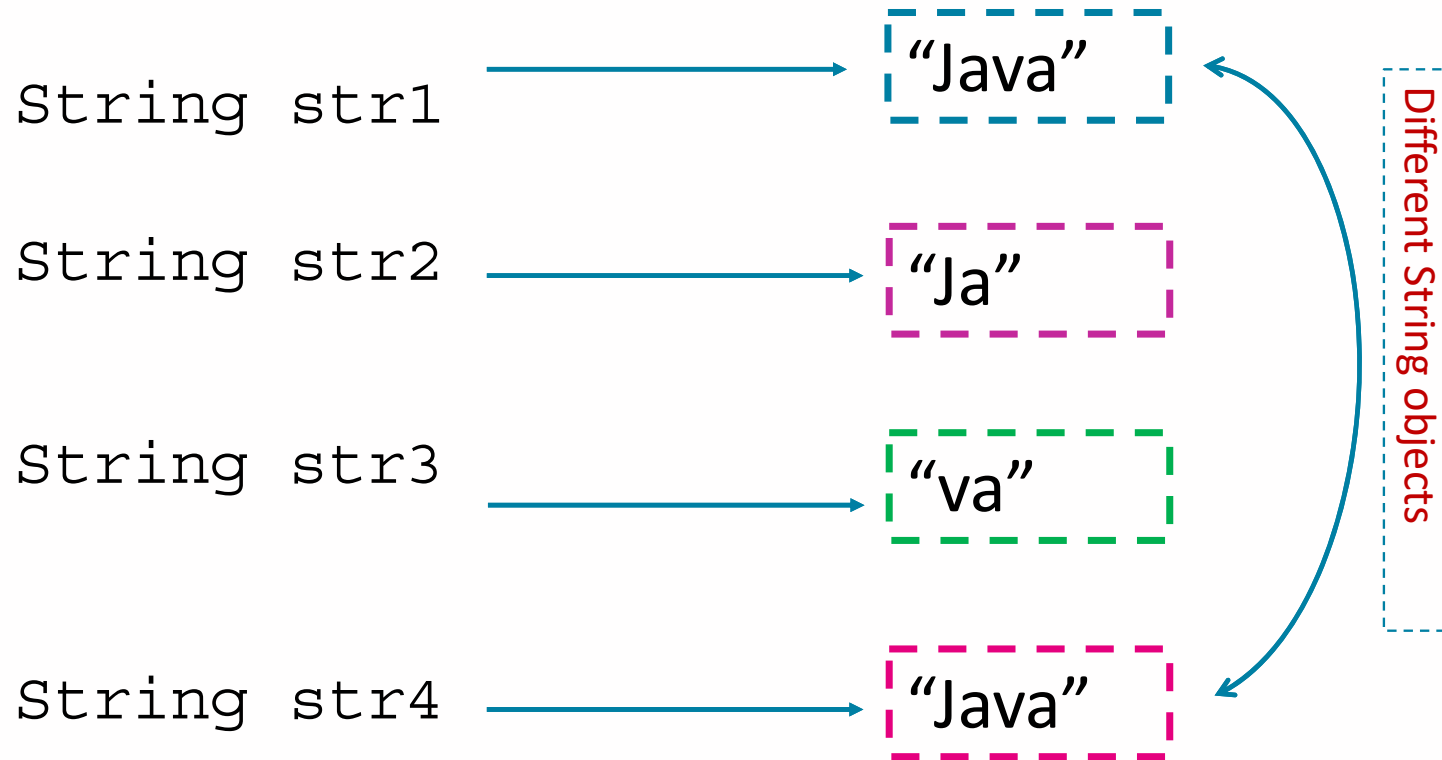
Why?

- Why is the value of **str1** not equal to the value of **str4**
- The answer is that the values of **str1** and **str4** are memory references to different objects
- It doesn't matter that the objects may contain the same data ("Java")
- When you use `==` with reference variables **you are simply checking if the variables point to the same object**



Variables

Objects



```
if(str1==str4){  
    System.out.println("true");  
} else{  
    System.out.println("false");  
}
```

The value of **str1** is the **memory location** where its String object is stored
The value of **str4** is the **memory location** where its String object is stored
So **str1** is not equal (==) to str4



Object Equality

- When checking for equality between objects you must use the **equals** method
- **The equals method is an instance method that all objects have**
- Its specific purpose is to define equality between objects
- It returns a **boolean** value



You can download this code snippet from Blackboard

```
StringEqualityDemo x
Compile Undo Cut Copy Paste Find... Close Source Code
*/
public class StringEqualityDemo
{
    /**
     * main method used to illustrate String equality
     *
     */
    public static void main (String[] args)
    {
        String str1 = "Java";
        String str2 = "Ja";
        String str3 = "va";
        String str4 = str2+str3;

        if(str1==str4){
            System.out.println("true");
        } else{
            System.out.println("false");
        }
    }
}
```



OLLSCOIL NA GAILLIMHĒ
UNIVERSITY OF GALWAY

Rewrite the code and run

```
String str1 = "Java";  
String str2 = "Ja";  
String str3 = "va";  
String str4 = str2+str3;
```

```
if(str1.equals(str4)){  
    System.out.println("true");  
} else{  
    System.out.println("false");  
}
```

Output:

true



In this case, we use the **equals** method of the String object referenced by **str1**

It accepts the value of **str4** as an input parameter and returns true or false

```
if(str1.equals(str4)) {  
    System.out.println("true");  
} else {  
    System.out.println("false");  
}
```



equals must be **commutative**

```
str1.equals(str4)
```

must return the same boolean value as...

```
str4.equals(str1)
```



Every object has an equals method

- **Every single object has an equals method**
- Because evaluating the equality between objects is a very common function
 - E.g for searching, sorting
- For the built-in classes of Java, the equals method will already be defined
- But for any class that you define **you will have to write the equals method**



Tutorial - Collections

- We will now spend a few minutes looking at the collection tutorial
 - There are two separate PDFs that can be found in Week 4 on Blackboard
- We will also look at looping over items in a collection



Grouping objects

Introduction to collections

Main concepts to be covered

- Collections
(especially **ArrayList**)

The requirement to group objects

- Many applications involve collections of objects:
 - Personal organizers.
 - Library catalogs.
 - Student-record systems.
- The number of items to be stored varies.
 - Items added.
 - Items deleted.

Java Class libraries

- Collections of useful classes.
- We don't have to write everything from scratch.
- Java calls its libraries, *packages*.
- Grouping objects is a recurring requirement.
 - The `java.util` package contains multiple classes for doing this.

An organizer for music files

- Single-track files may be added.
- There is no pre-defined limit to the number of files/tracks.
- It will tell how many file names are stored in the collection.
- It will list individual file names.
- It will allow you to remove a file

v1

- One class : Music Organizer
- We will use Strings as Files for version 1
- Methods:
 - addFile
 - getNumberOfFiles
 - listFile
 - removeFile

Collections

- We specify:
 - the type of collection: **ArrayList**
 - the type of objects it will contain:
<String>
 - **private ArrayList<String> files;**
- We say, “ArrayList of String”.

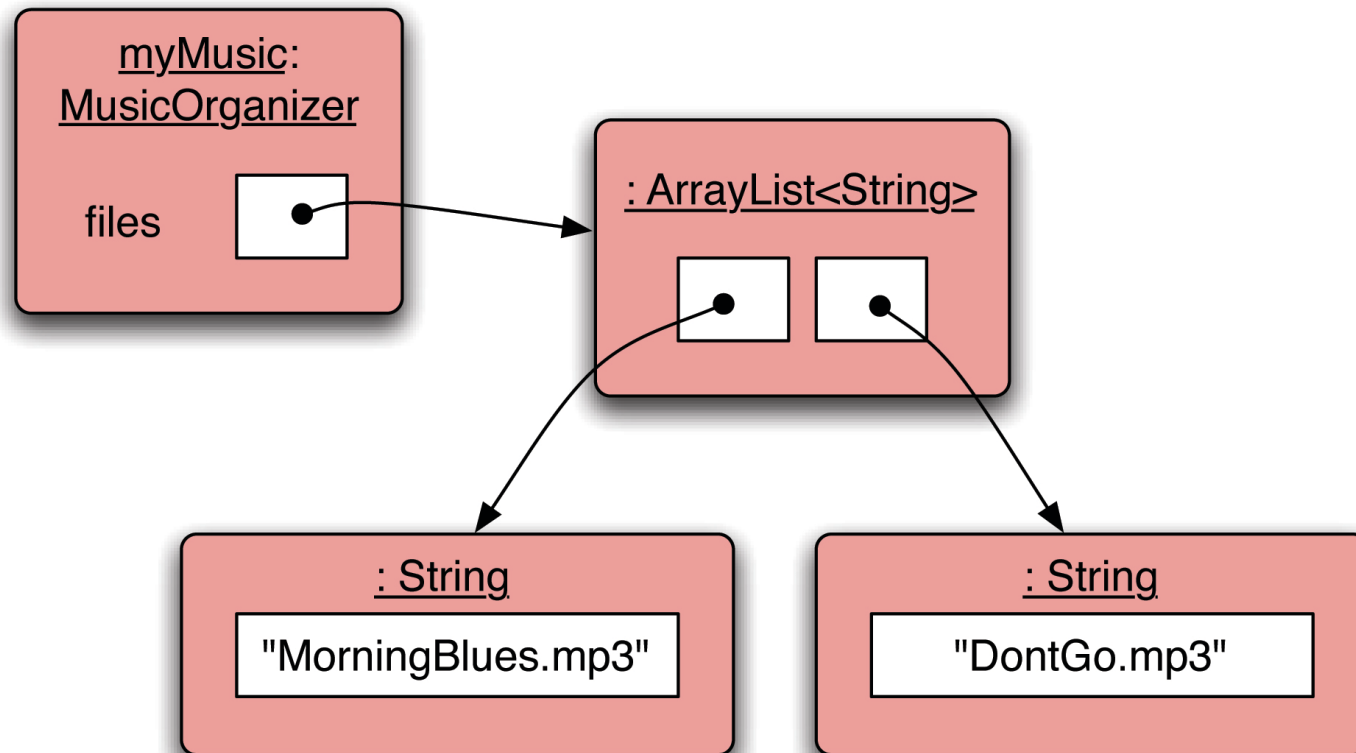
Generic classes

- Collections are known as *parameterized* or *generic* types.
- **ArrayList** implements list functionality:
 - **add**, **get**, **size**, etc.
- The type parameter says what we want a list of:
 - **ArrayList<Person>**
 - **ArrayList<TicketMachine>**
 - etc.

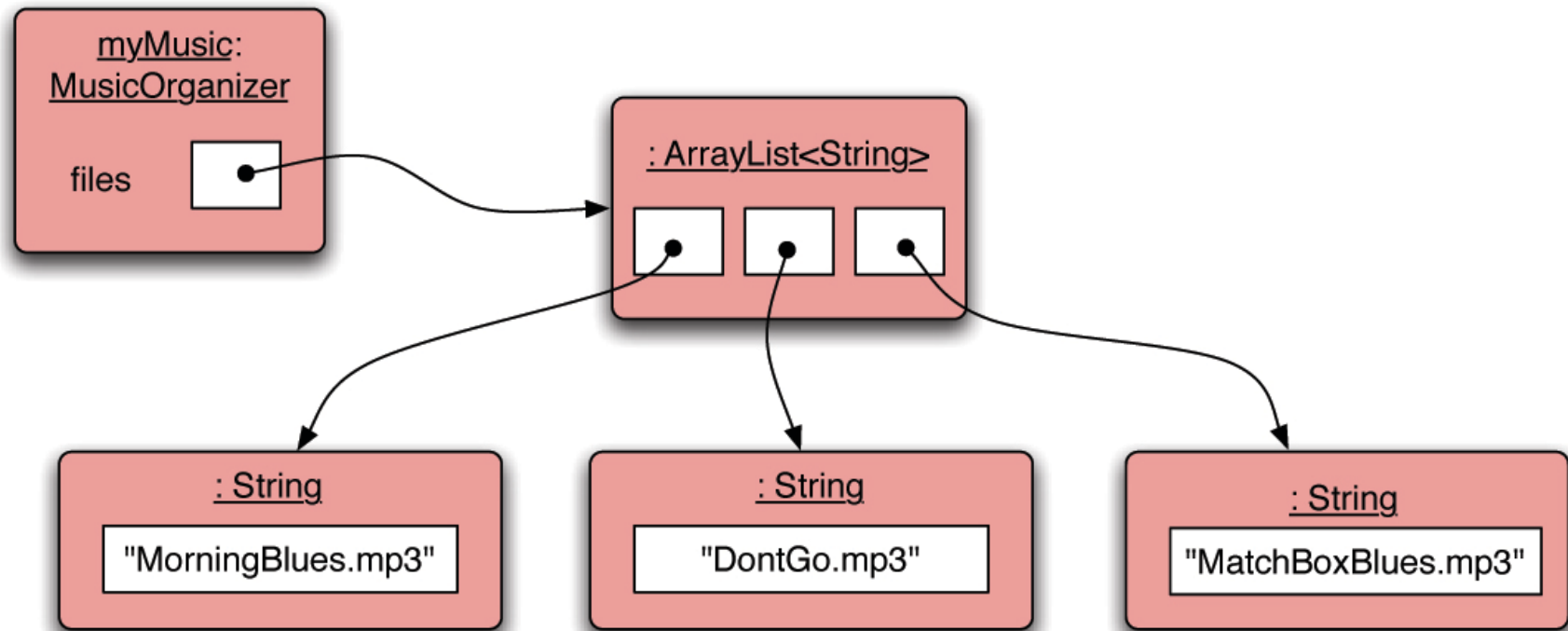
Creating an ArrayList object

- In versions of Java prior to version 7:
 - `files = new ArrayList<String>();`
- Java 7 introduced ‘diamond notation’
 - `files = new ArrayList<>();`
- The type parameter can be inferred from the variable being assigned to.
 - A convenience we will use.

Object structures with collections



Adding a third file



Features of the collection

- It increases its capacity as necessary.
- It keeps a private count:
 - `size()` accessor.
- It keeps the objects in order.
- Details of how all this is done are hidden.
 - Does that matter? Does not knowing how prevent us from using it?
- No - this is a key idea of encapsulation₁₂

Generic classes

- We can use `ArrayList` with any class type:

`ArrayList<TicketMachine>`

`ArrayList<ClockDisplay>`

`ArrayList<Track>`

`ArrayList<Person>`

- Each will store multiple objects of the specific type.

Using the collection

```
public class MusicOrganizer
{
    private ArrayList<String> files;

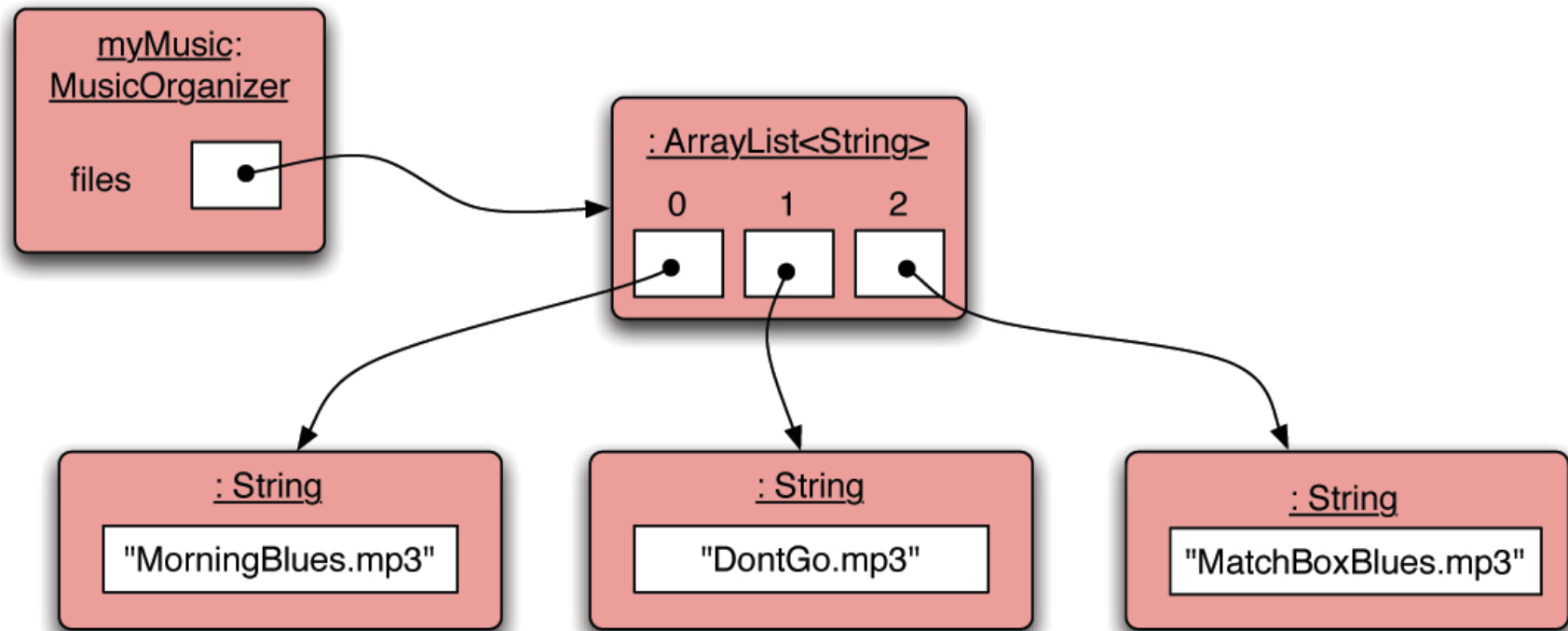
    ...

    public void addFile(String filename)
    {
        files.add(filename); ← Adding a new file
    }

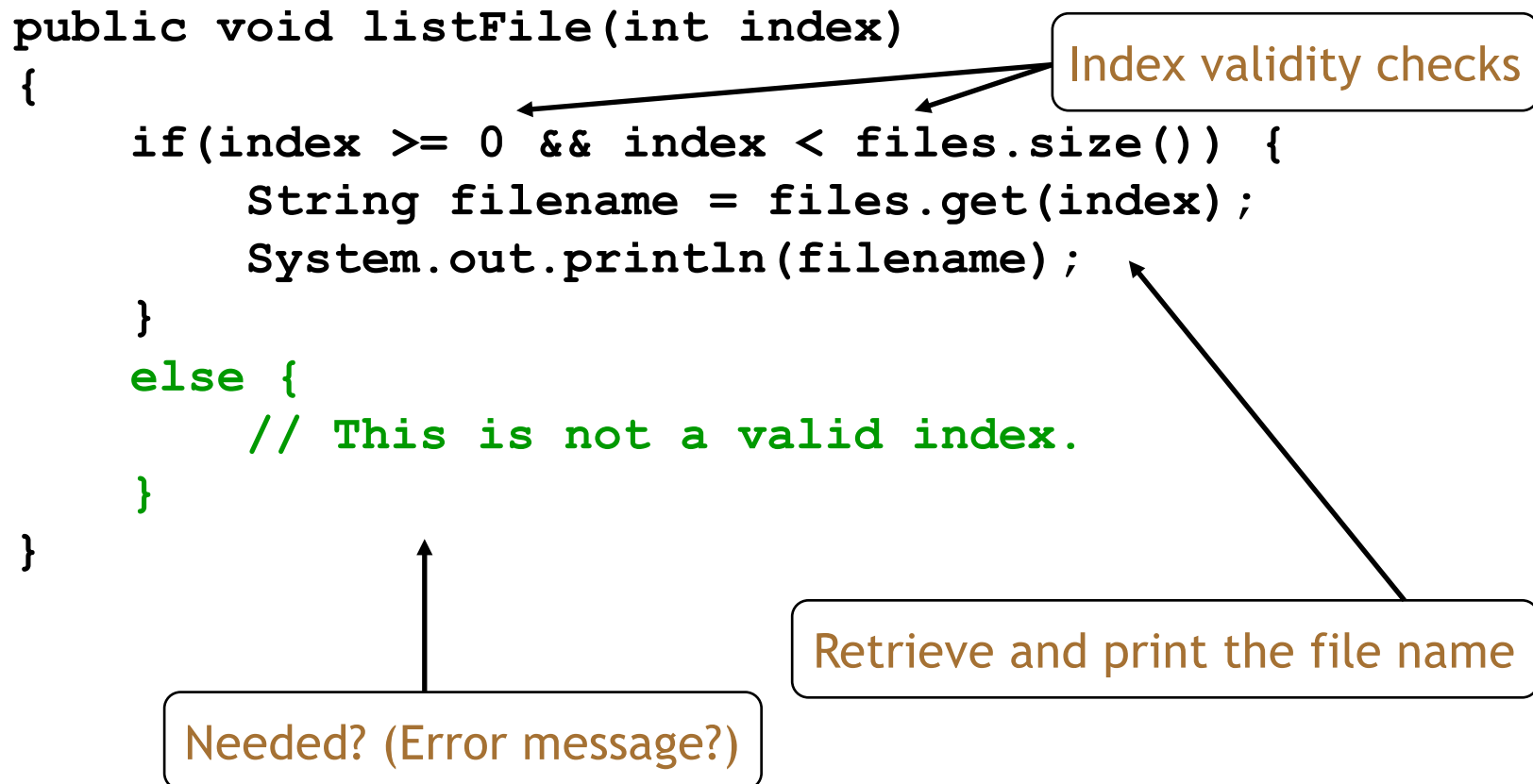
    public int getNumberOfFiles()
    {
        return files.size(); ← Returning the number of files
                               (delegation)
    }

    ...
}
```

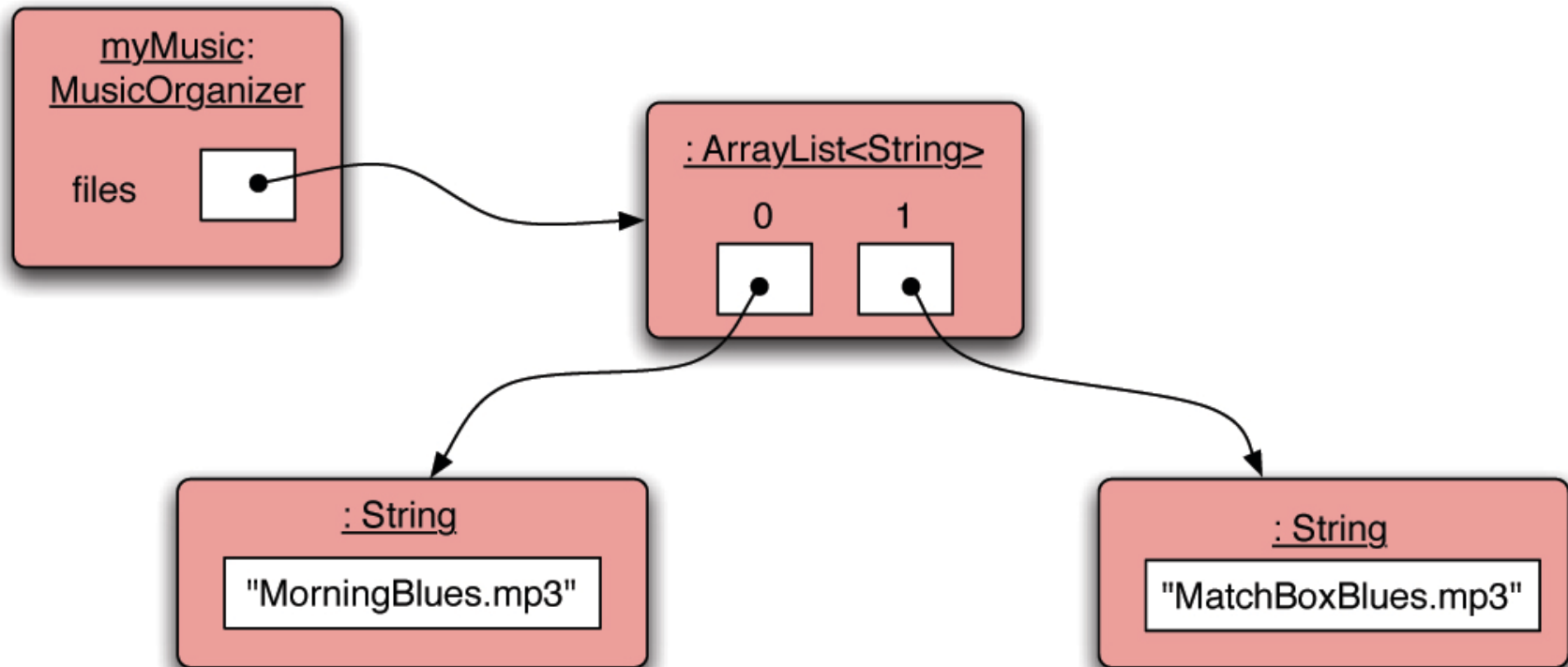
Index numbering



Retrieving from the collection



Removal may affect numbering



The general utility of indices

- Using integers to index collections has a general utility:
 - ‘next’ is: `index + 1`
 - ‘previous’ is: `index - 1`
 - ‘last’ is: `list.size() - 1`
 - ‘the first three’ is: the items at indices 0, 1, 2
- We could also think about accessing items in sequence: 0, 1, 2, ...

Review

- Collections allow an arbitrary number of objects to be stored.
- Class libraries usually contain tried-and-tested collection classes.
- Java's class libraries are called *packages*.
- We have used the `ArrayList` class from the `java.util` package.

Review

- Items may be added and removed.
- Each item has an index.
- Index values may change if items are removed (or further items added).
- The main **ArrayList** methods are **add**, **get**, **remove** and **size**.
- **ArrayList** is a *parameterized* or *generic* type.

Learning task

Create a class that can organise a group of objects

- E.g. a Library of books
- A course with students registered
- A Team with players
- A league with teams

The choice is yours

Grouping objects

the for-each and while loops

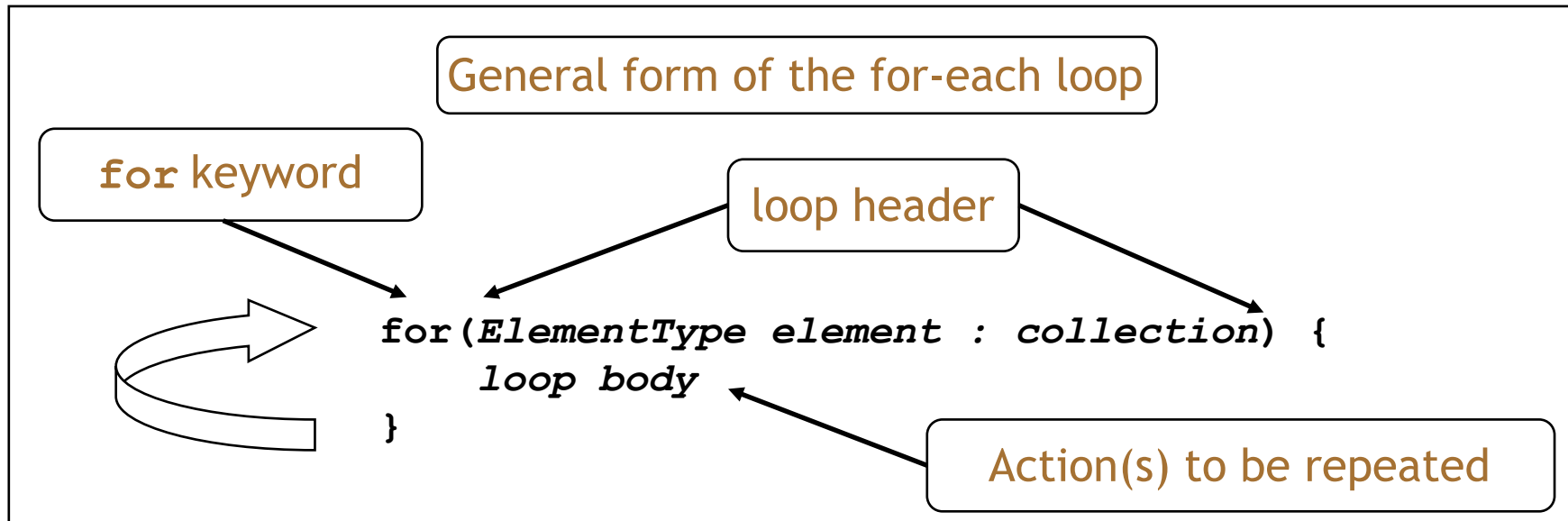
Iteration

- We often want to perform some actions an arbitrary number of times.
 - E.g., print all the file names in the organizer. How many are there?
- Most programming languages include *loop statements* to make this possible.
- Java has several sorts of loop statement.
 - We will start with its *for-each loop*.

Iteration fundamentals

- The process of repeating some actions over and over.
- Loops provide us with a way to control how many times we repeat those actions.
- **With a collection, we often want to repeat the actions: *exactly once for every object in the collection.***

For-each loop pseudo code



Pseudo-code expression of the operation of a for-each loop

Using each *element* in *collection* in order, do the things in the *loop body* with that *element*.

A Java example

```
/**
 * List all file names in the organizer.
 */
public void listAllFiles()
{
    for(String filename : files) {
        System.out.println(filename);
    }
}
```

Using each *filename* in *files* in order, print *filename*

Review

- Loop statements allow a block of statements to be repeated.
- The for-each loop allows iteration over a whole collection.
- With a for-each loop *every* object in the collection is made available *exactly once* to the loop's body.

Selective processing

- Statements can be nested, giving greater selectivity to the actions:

```
public void findFiles(String searchString)
{
    for(String filename : files) {
        if(filename.contains(searchString)) {
            System.out.println(filename);
        }
    }
}
```

contains gives a partial match of the filename;
use equals for an exact match

break

- What if we wanted to stop searching immediately after we find the first match?
- `break`

Selective processing

- Statements can be nested, giving greater selectivity to the actions:

```
public void findFiles(String searchString)
{
    for(String filename : files) {
        if(filename.contains(searchString)) {
            System.out.println(filename);
            break;
        }
    }
}
```

breaks out of the loop;



Critique of for-each

- Easy to write.
- Termination happens naturally.
- *The collection cannot be changed by the actions* (e.g. can't remove an element)
- There is no index provided.
 - Not all collections are index-based.
- We can stop part way using the **break** keyword.
- It provides 'definite iteration' - aka 'bounded iteration' .

Grouping objects

Indefinite iteration - the while loop

Main concepts to be covered

- The difference between definite and indefinite (unbounded) iteration.
- Loops: the while loop

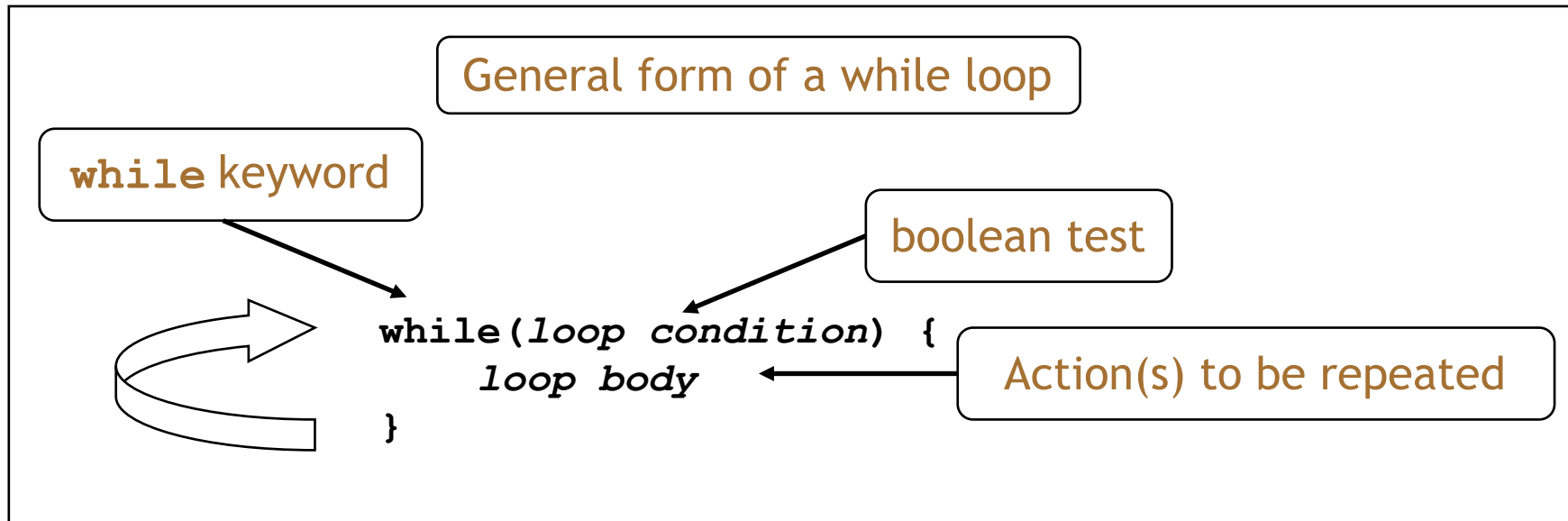
While loop

- A for-each loop repeats the loop body for every object in a collection.
- You use a while loop when you want to **keep iterating until a certain condition is met**
- **This is indefinite (unbounded) iteration**
- Beware - if the condition isn't met then you will have have an infinite loop

The while loop

- We use a boolean condition to decide whether or not to keep iterating.
- This is a *very* flexible approach. Termination of the loop depends on the condition
- Not just tied to collections.

While loop pseudo code



“while we wish to continue, do the things in the loop body”

Search

- What if we want to search for a filename and we want to return the the **index** of the first element that matches our input
- Remember, the for-each loop doesn't have an index as part of its syntax

We keep searching until

- *Either* there are no more items to check:
`index >= files.size()`
- *Or* the item has been found:
`found == true`

```
public int findFile(String searchString) {

    int index = 0;
    boolean found = false;
    while(index < files.size() && !found) {
        String file = files.get(index);
        if(file.contains(searchString)) {
            found = true;
            return index; // We don't need to keep looking.
        }
        else {
            index++;
        }
    }
    return -1; // if we get this far, the item has not been
    found
}
```

for-each versus while

- for-each:
 - easier to write.
 - safer: it is guaranteed to stop.
- while:
 - we don't *have to* process the whole collection.
 - doesn't even have to be used with a collection.
 - take care: could create an *infinite loop*.

Learning exercise

- Write the previous method using a **while loop and a conventional for loop** so that it prints out the first 3 matches of the searchString
- Once it encounters 3 matches it can exit the loop



OLLSCOIL NA GAILLIMHE
UNIVERSITY OF GALWAY

CT2106

Object Oriented Programming



Dr. Frank Glavin
Room 404, IT Building
Frank.Glavin@UniversityofGalway.ie
School of Computer Science

University
ofGalway.ie

Key idea in a class hierarchy

- The top of the hierarchy represents the **most generic attributes and behaviours**
- The bottom (the leaves) represent the **most specific attributes and behaviours**
- **Each level inherits and customises the attributes and behaviours from the level above it**



OOP Inheritance

The means by which objects automatically receive features (fields) and behaviours (methods) from their super classes



Java class hierarchy

- At the top of the Java class hierarchy is a class called `java.lang.Object`
- All classes inherit *implicitly* from `java.lang.Object`
- This means that a class doesn't have to specify explicitly that `java.lang.Object` is its superclass



Revision

We are used to reference type declarations like this

```
Bicycle bike = new Bicycle(2,14);  
String strng1 = "Hello";  
String strng1 = new String("Hello");
```

i.e. the variable type matches the object type;



Rules of class Hierarchy

- In Java, the variable type can be the superclass of the object

```
Object obj = new Bicycle(2,14);
```

```
Object object1 = "Hello";
```

```
Object object2 = new String("Hello");
```

- The variable type can be **any superclass** of the object, not just `java.lang.Object`

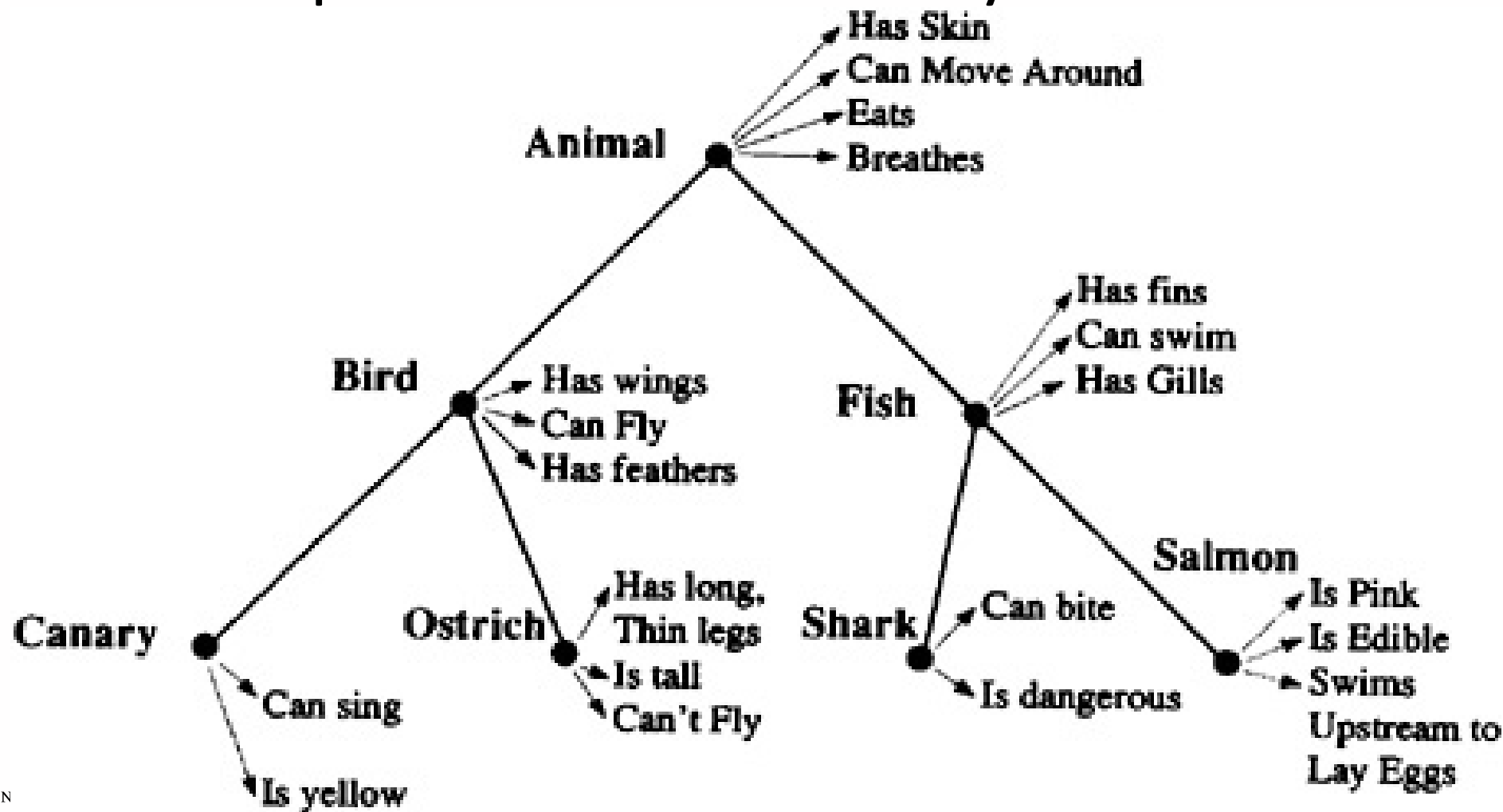


Explicit Inheritance

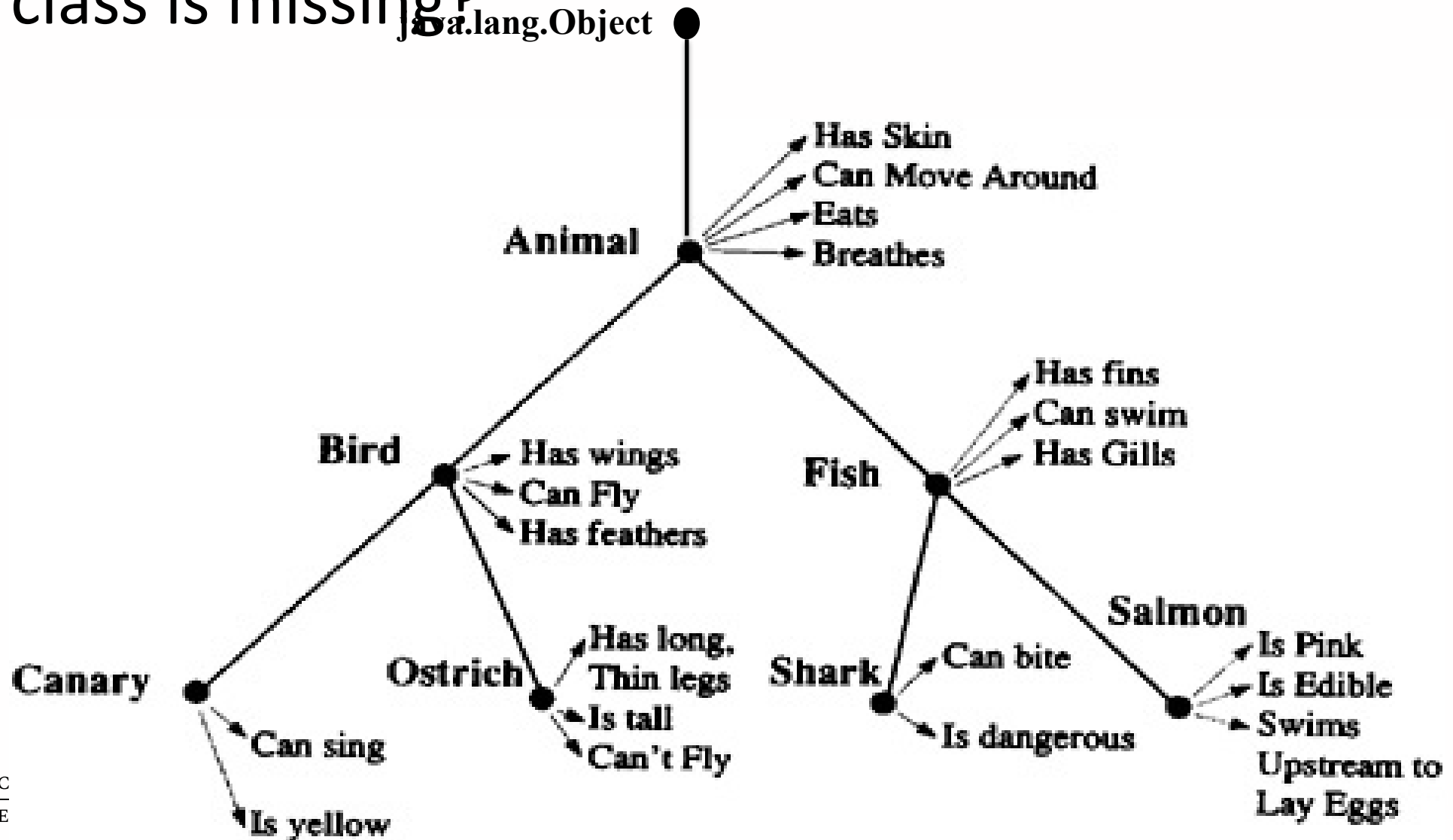
- All classes inherit methods *implicitly* from `java.lang.Object`
- In other words you don't have to tell Java that a class inherits from `java.lang.Object`
- Two common methods inherited from `java.lang.Object` ?
 - `equals()`
 - `toString()`
- In every other case, you have to tell Java which classes are in a superclass relationship



Assignment 3: Implement this hierarchy



What class is missing?

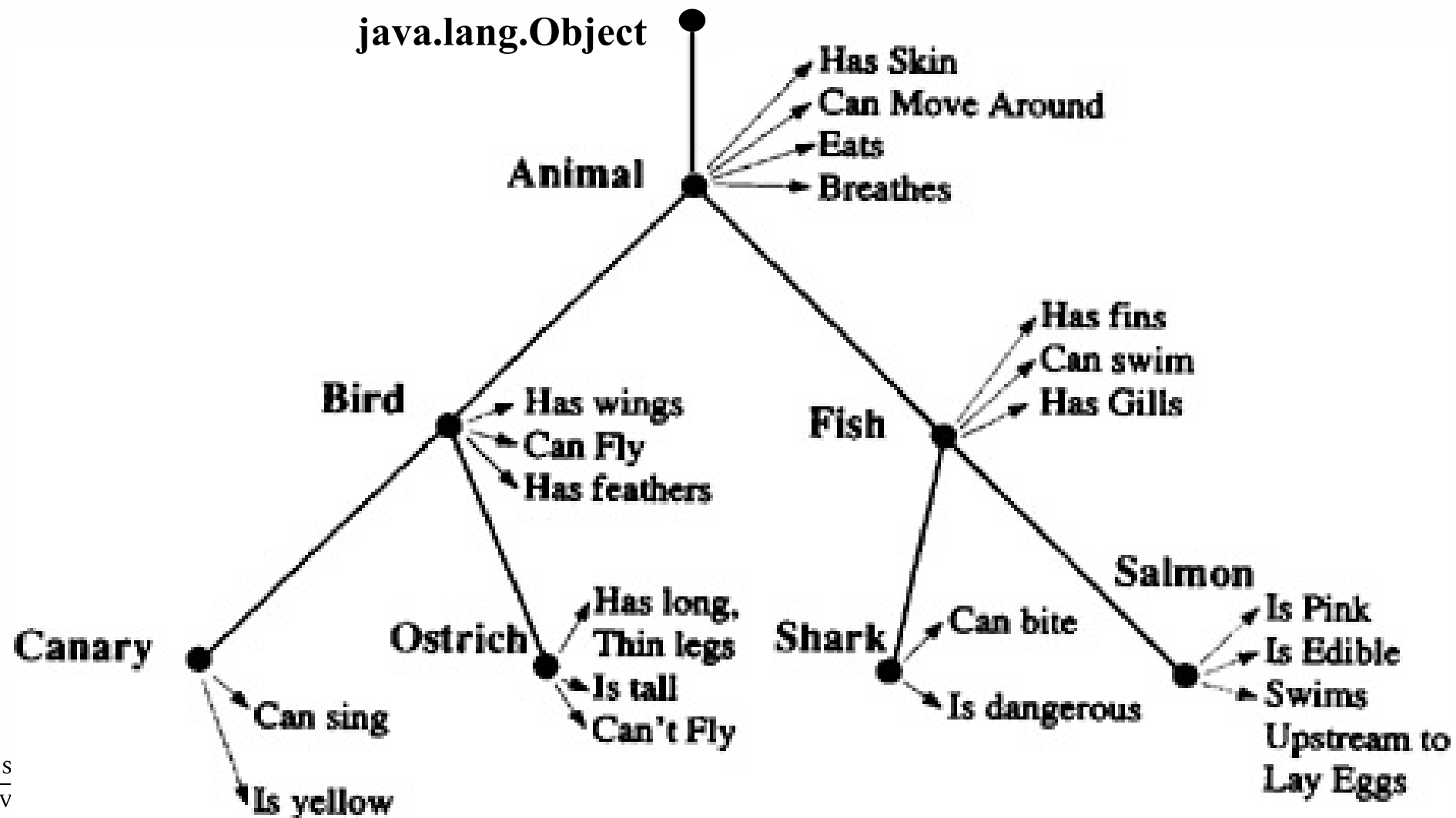


Inheritance

- The **Canary** Type **inherits** features from the **Bird** Type and the Bird Type **inherits** features from the **Animal** Type. The Animal Type **inherits** from **java.lang.Object**
- The Canary **adds** its own features (*yellow, sings*) to the features inherited from the Bird type
- The Bird Type **adds** its own features (*feathers, wings*) and **adapts** a feature from the Animal type (*move - > fly*)



Fields or Methods



Fields or Methods?

- Some properties are definitely fields (hasSkin, hasFeathers)
- Which are methods ?
- The decision will be helped by the context of the application
- Let's say that these classes are part of a game, where animal avatars have certain behaviours
 - Move
 - Eating
 - Making noise
- Now the decision is easy



Steps

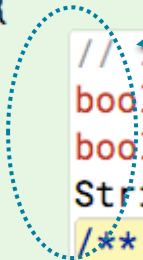
1. Create the classes - lets start with the left hand side of the tree
2. Insert the inheritance relationships
3. Insert the fields
4. Insert the methods
5. Override necessary fields
6. Override necessary methods
7. Test by putting objects in an array and calling their behaviours



```
/**
 * Write a description of class Animal here.
 *
 * @author (conor hayes)
 * @version (October 5th 2017)
 */
```

Don't make the fields private if you want them to be inherited

```
public class Animal
{
```



```
// instance variables - replace the example below with your own
boolean hasSkin;
boolean breathes;
String colour;
```

```
/**
 * Constructor for objects of class Animal
 */
```

```
public Animal()
{
```

```
    breathes = true; //all the subclasses of Animal inherit this property and value
    hasSkin = true; // all the subclasses of Animal inherit this property and value
    colour = "grey"; //all the subclasses of Animal inherit this property and value
}
```

```
/**
 * move method
 * param int distance - the distance the Animal should move
 * All subclasses inherit this method
 */
```

```
public void move(int distance){
    System.out.printf("I move %d metres \n", distance);
}
```



```
/**
 * Write a description of class Bird here.
 *
 * @author (conor hayes)
 * @version (October 5th 2017)
 */
public class Bird extends Animal
{
    //instance variables (fields)
    boolean hasFeathers;
    boolean hasWings;
    boolean flies;

    /**
     * Constructor for objects of class Bird
     */
    public Bird()
    {
        super(); //calls the constructor of its superclass - Animal.
        colour = "black"; //overrides the value of colour inherited from Animal
        hasFeathers = true; //all the subclasses of Bird inherit this property and value
        hasWings = true; //all the subclasses of Bird inherit this property and value
        flies = true; //all the subclasses of Bird inherit this property and value
    }
}
```



extends indicates the subclass to be extended (inherited from)

You must call the constructor of the superclass using the method call **super()**

If the superclass constructor takes a parameter then the call to super must include a value of the parameter. E.g. **super("joey")**



```

public class Canary extends Bird
{
    // instance variables - replace the example below with your own
    String name; // the name of this Canary

    /**
     * Constructor for objects of class Canary
     */
    public Canary(String name)
    {
        super(); // call the constructor of the superclass Bird
        this.name = name;
        colour = "yellow"; // this overrides the value inherited from Bird
    }

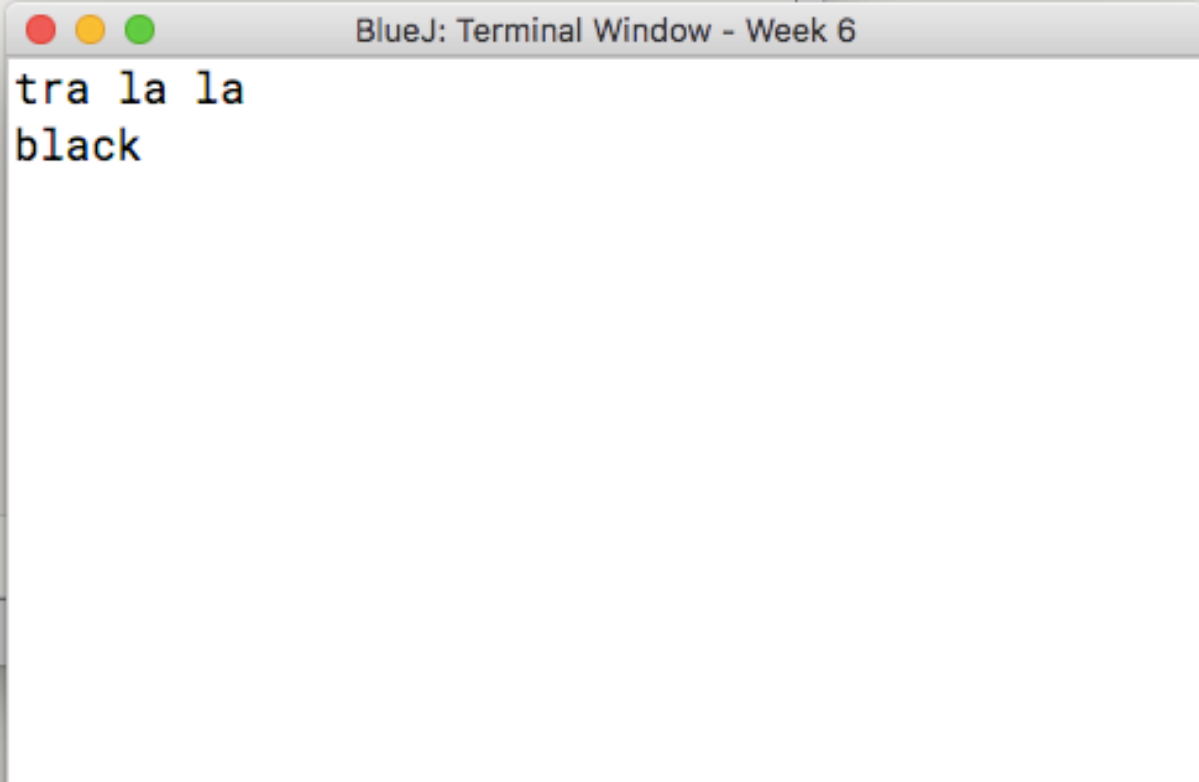
    /**
     * Sing method overrides the sing method
     * that was inherited from superclass Bird
     */
    @Override // good programming practice to use @Override to denote overridden methods
    public void sing(){
        System.out.println("tweet tweet tweet");
    }
}

```



Code pad

```
Bird bird = new Bird();  
bird.sing();  
System.out.println(bird.getColour());
```



BlueJ: Terminal Window - Week 6

```
tra la la  
black
```



Code pad

11

```
Canary john = new Canary("John");
john.sing();
System.out.println(john.getColour());
```

BlueJ: Terminal Window - Week 6

```
tweet tweet tweet
yellow
```



- Sing method in Canary overrides the Sing method inherited from Bird
- Canary overrides the value of the colour field inherited from Bird. Bird objects are black. Canary objects are yellow



Abstract

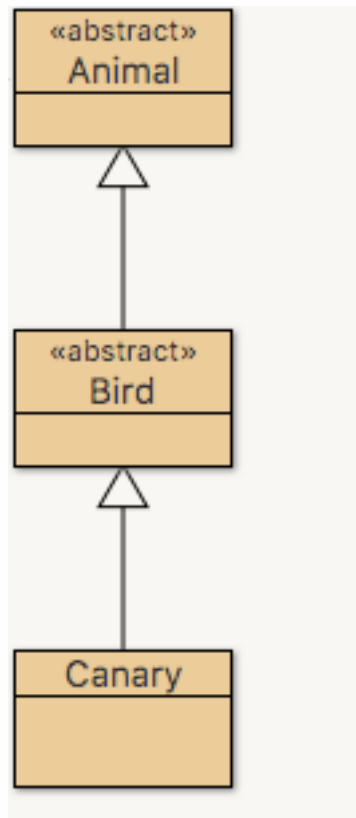
- It may not make sense to have an object of type superclass
- E.g. have you ever seen an an Animal or Bird object walking about
- Java allows you to specify which classes can be made into objects
- And which are used just for inheritance purposes

```
public abstract class Animal  
{
```

```
public abstract class Bird extends Animal  
{
```



Adding the word **abstract** to the class definition tells Java that it can't make objects from this class



```
Animal animal = new Animal();
    Error: Animal is abstract; cannot be instantiated
Bird bird = new Bird();
    Error: Bird is abstract; cannot be instantiated
```



Code pad example

- However an abstract class can still be used as the type of a reference variable

```
Bird bird = new Canary("John");  
Animal animal = new Canary("Mary")
```

```
Animal animal = new Animal();  
    Error: Animal is abstract; cannot be instantiated  
Bird bird = new Bird();  
    Error: Bird is abstract; cannot be instantiated  
Bird bird = new Canary("John");  
Animal animal = new Canary("Mary");
```



Key points to remember

1. You must explicitly invoke the constructor method of the superclass using `super()` or `super(params)`;
2. Private fields or methods are not inheritable
3. A subclass inherits the fields and field values of the superclass
4. A subclass can override any fields or methods inherited from the superclass
5. The *abstract* keyword can be used to designate classes that can only be extended
6. An abstract class can still be used to as the type of a reference variable





OLLSCOIL NA GAILLIMHE
UNIVERSITY OF GALWAY

CT2106

Object Oriented Programming



Dr. Frank Glavin
Room 404, IT Building
Frank.Glavin@UniversityofGalway.ie
School of Computer Science

University
ofGalway.ie

Lecture Topics

- Abstract classes and methods
- Polymorphism



OLLSCOIL NA GAILLIMHÉ
UNIVERSITY OF GALWAY

Abstract

- It may not make sense to have an object of type superclass. E.g. Animal or Bird
- E.g. have you ever seen an Animal or Bird object walking/flying about?
- You've seen *specific* types of Animals and specific types of Birds
- Animal and Birds are **abstractions**



Abstraction

1. variable noun

An **abstraction** is a general idea rather than one relating to a particular object, person, or situation.

<https://www.collinsdictionary.com/dictionary/english/abstraction>



OLLSCOIL NA GAILLIMHÉ
UNIVERSITY OF GALWAY

Abstract Keyword

- You can declare a class to be **abstract**

```
public abstract class Animal  
{
```

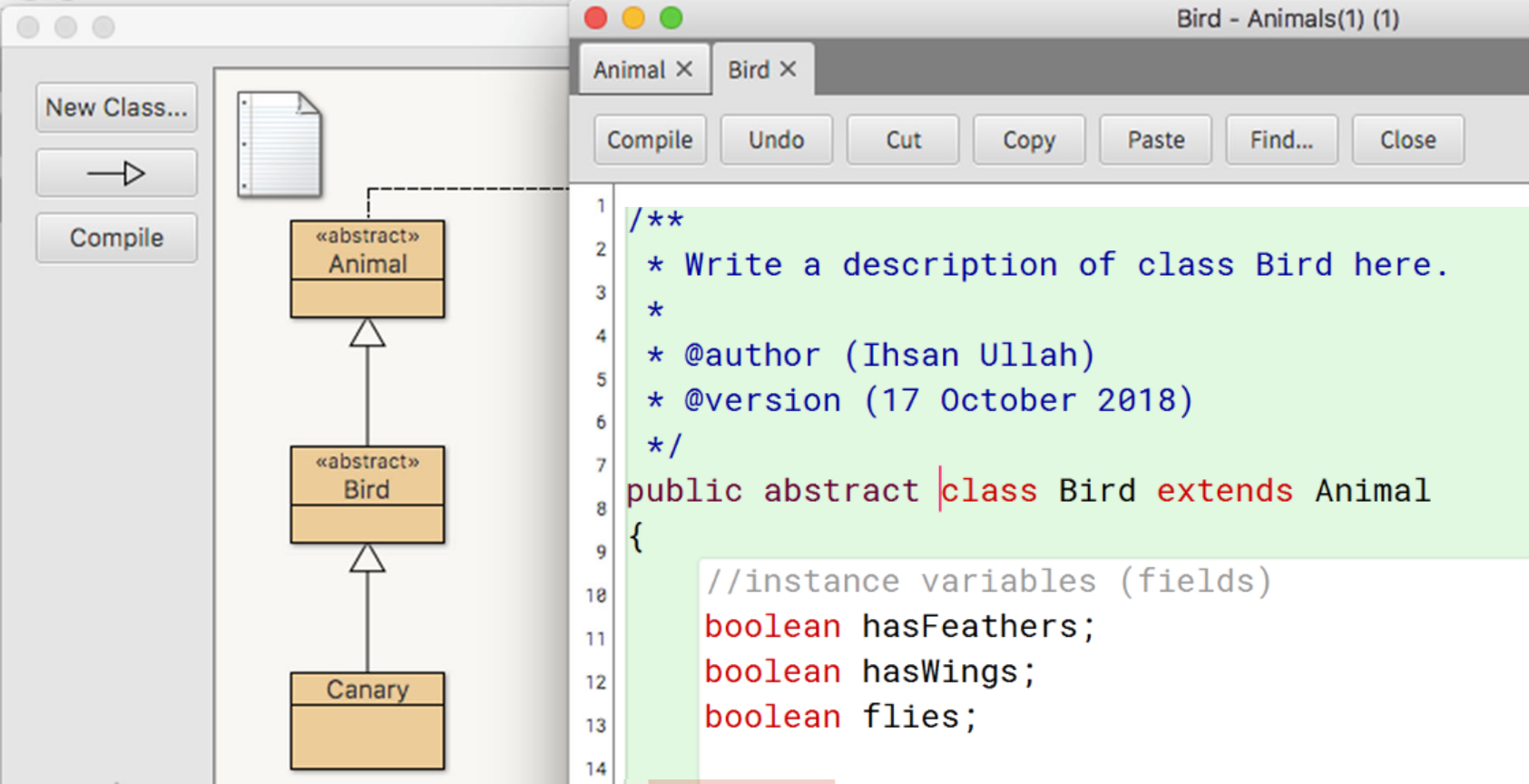
```
public abstract class Bird extends Animal  
{
```

- Java allows you to specify which classes can be made into objects
- ..and which are **abstract** and used just for inheritance purposes



Code

In BlueJ
Make the
Animal and
Bird classes
abstract



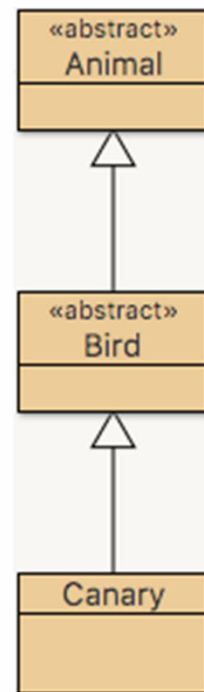
The screenshot shows the BlueJ IDE interface. On the left, the class browser displays a hierarchy: «abstract» Animal (with a dashed line indicating it is abstract), «abstract» Bird (with a dashed line indicating it is abstract), and Canary. On the right, the code editor shows the following Java code for the Bird class:

```
1 /**
2  * Write a description of class Bird here.
3  *
4  * @author (Ihsan Ullah)
5  * @version (17 October 2018)
6  */
7 public abstract class Bird extends Animal
8 {
9     //instance variables (fields)
10     boolean hasFeathers;
11     boolean hasWings;
12     boolean flies;
13 }
14
```



abstract Keyword

Adding the word **abstract** to the class definition tells Java that it can't make objects from this class
Now, as you did before, try to create an Animal and Bird object



```
Animal animal = new Animal();
    Error: Animal is abstract; cannot be instantiated
Bird bird = new Bird();
    Error: Bird is abstract; cannot be instantiated
```



abstract

- First effect is that you no longer can create objects from the abstract class
- However, all the existing rules of inheritance still apply

```
public abstract class Bird extends Animal
{
    //instance variables (fields)
    boolean hasFeathers;
    boolean hasWings;
    boolean flies;

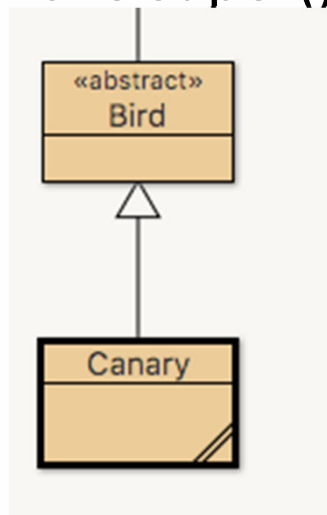
    /**
     * Constructor for objects of class Bird
     */
    public Bird()
    {
        super(); //calls the constructor of its
        colour = "black"; //overrides the value
        hasFeathers = true; //all the subclasses
        hasWings = true; //all the subclasses of
        flies = true; //all the subclasses of B
    }
}
```

- Sub-classes of Bird inherit its non-private fields



abstract

Even though Bird is declared as an abstract class a subclass (e.g. Canary) still has to invoke super()

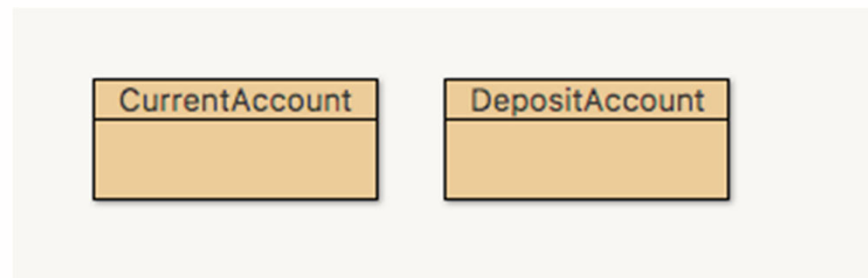


```
/**
 * Constructor for objects of class Canary
 */
public Canary(String name)
{
    super(); // call the constructor of the superclass Bird
    this.name = name;
    colour = "yellow"; // this overrides the value inherited from Bird
}
```



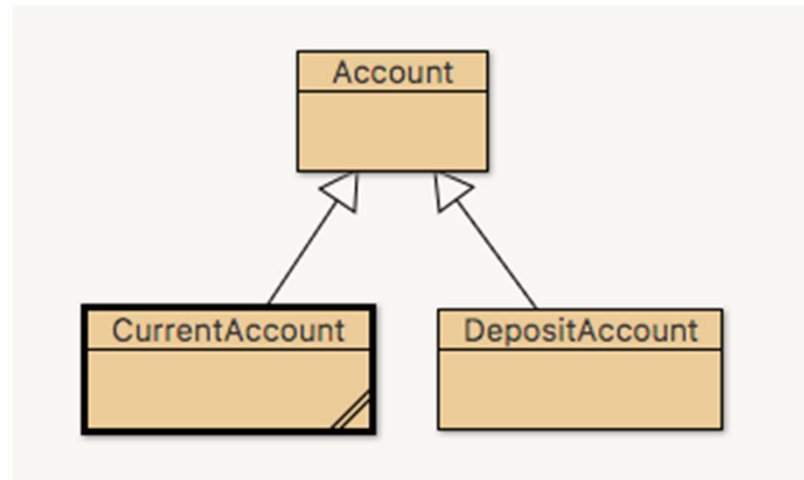
Why use an abstract class?

- In situations where you want to use inheritance but do not want another developer to create an object from the superclass.
- E.g a banking app has two bank account types :
- Current Account and Deposit Account



Why use abstract

- Both account types share many of the same fields and methods
- So the developer creates a superclass, Account, to hold all the shared fields and methods



Why use abstract

- However a trainee developer then writes the following line of code

```
Account account = new Account();
```

- This is a problem as there is no such thing in the Banking app as an Account.
- An account must either be a Current Account or a Deposit Account



To prevent this happening, the senior developer declares the Account class abstract

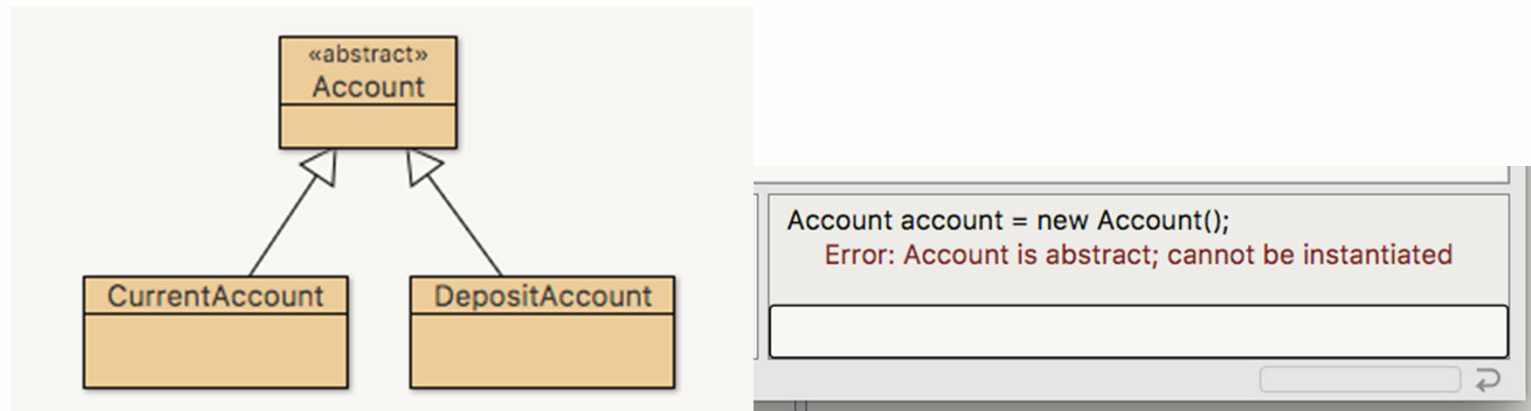
```
public abstract class Account
```



Why use abstract

As before, CurrentAccount and DepositAccount still inherit fields and methods from the abstract Account class

But Account itself cannot be *instantiated* (an object cannot be made of it)



Methods in an abstract class

As you've seen, an abstract class can have standard methods
These methods are inherited automatically by the subclass

```
/**
 * move method
 * param int distance - the distance the Animal should move
 * All subclasses inherit this method
 */
public void move(int distance){
    System.out.printf("I move %d metres \n", distance);
}
```



Methods in an abstract class

As we've seen, a subclass can **override** (provide their own specific implementation) of the inherited methods

```
/**
 * the move method in Bird overrides the move method
 * inherited from superclass Animal
 */
@Override // good programming practice to use @Override to denote overridden methods
public void move(int distance){
    if(flies){
        System.out.printf("I fly %d metres \n", distance);
    }else{
        System.out.printf("I am a bird but cannot fly. I walk %d metres \n", distance);
    }
}
```



e.g. this is the overridden move method in the Bird class

Abstract methods

- Abstract classes can also have **abstract methods**
- Abstract methods are methods **with no body**

E.g. `public abstract void sing();`

- In other words, they do nothing
- So what are abstract methods used for?



Demonstration

- Open up the *Animal* class in BlueJ
- Go to the *move* method

```
public void move(int distance){  
    System.out.printf("I move %d metres \n", distance);  
}
```

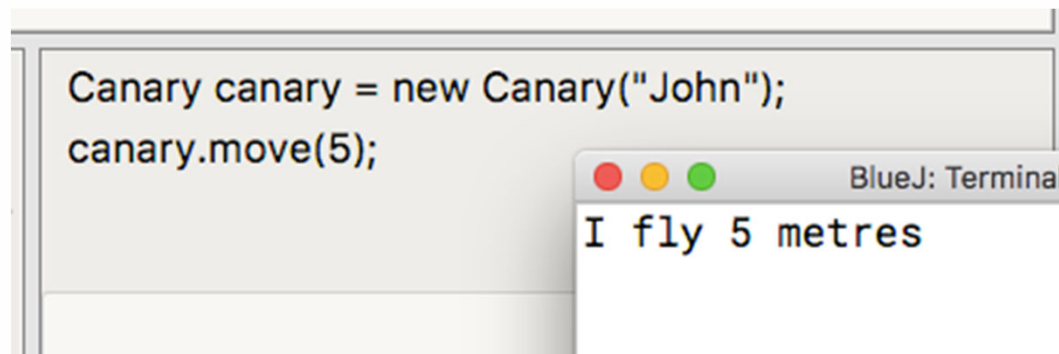
- Make it an **abstract method**
- This involves removing its body and simply keeping the method signature followed by a ‘;’
- Now compile the full project

```
public abstract void move(int distance);
```



Demonstration

- Your code still compiles
- In code pad, type the the following (hit return after each line)



```
Canary canary = new Canary("John");
canary.move(5);
```

BlueJ: Terminal
I fly 5 metres

- Where is the move functionality coming from?
- **From Bird's move method**



Demonstration

Canary's move functionality comes from Bird

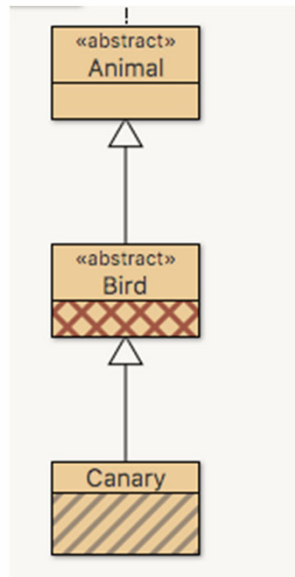
Now delete (or comment out) the move method from Animal

```
/**
 * move method
 * param int distance - the distance the Ani
 * All subclasses inherit this method
 */
//public abstract void move(int distance);
```

Recompile your project



Now Bird won't compile
Check what the error is
So what is the role of *move* in Animal ?



As an abstract method, it provides the **definition** of a method that at least one of its subclasses **must implement**



The meaning of the the abstract method *move* in the Animal class:

*“All animals **must move**, but it is up to each specific animal to decide how it moves”*



Concrete

- The adjective **concrete** is often used in OOP to denote a class or method that is not abstract
- i.e. The class or method is fully implemented
- In our example, Canary is a concrete class
- The move method in Bird is a concrete method



Reference Type

An abstract class is **often** used as the type of a reference variable

Try this in code pad

```
Bird bird = new Canary("John");  
Animal animal = new Canary("Mary");
```

Here we have two concrete objects referenced by variables whose type is an abstract class
Very common approach in OOP



abstract class and method summary

- The **abstract** keyword allows you to represent a class that should not be instantiated (made an object of)
- Inheritance from the abstract class happens the same as before
- An abstract class may have concrete and **abstract methods**
- An an abstract method does not have a method body
- It is there to provide a definition of a method that at least one of its subclasses must implement (make concrete)
- In our case – having an abstract method move is like saying “All animals **must move**, but it is up to each animal to decide how it moves”



Polymorphism



Polymorphism

- **Polymorphism** (from Greek *polys*, "many, much" and *morphē*, "form, shape")
- Polymorphism refers to how an object can be treated as belonging to several types as long as those types **are higher** than the object's type in the class hierarchy
- Thus, In the code snippet below, a Canary can be treated as a **Bird** type and as an **Animal** type

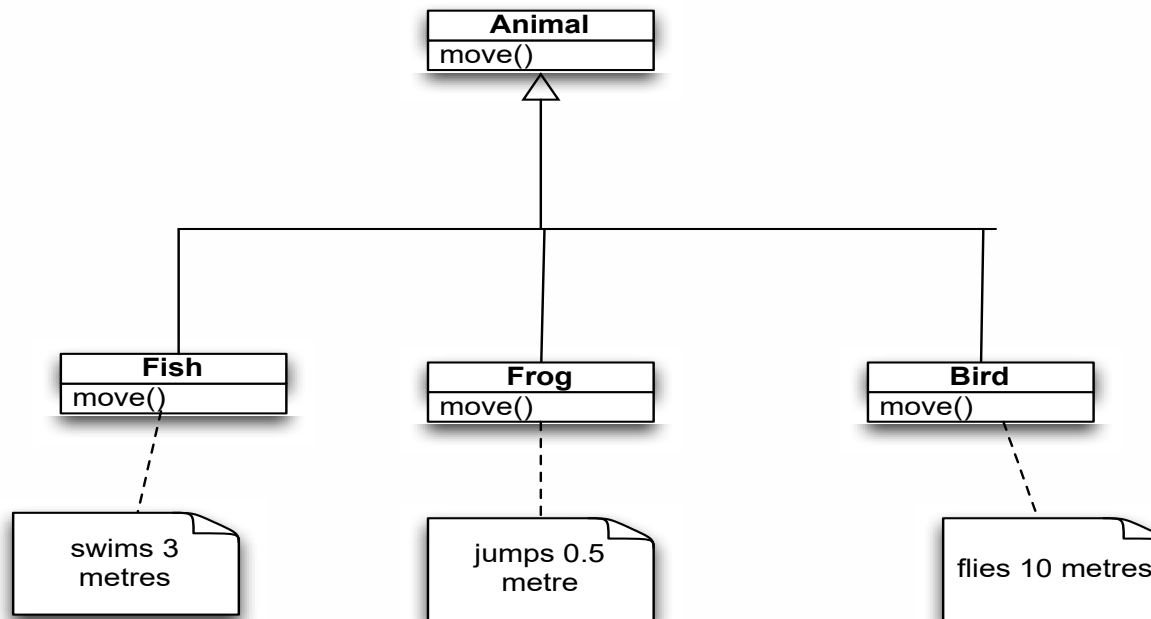
```
Bird bird = new Canary("John");  
Animal animal = new Canary("Mary");
```



Example

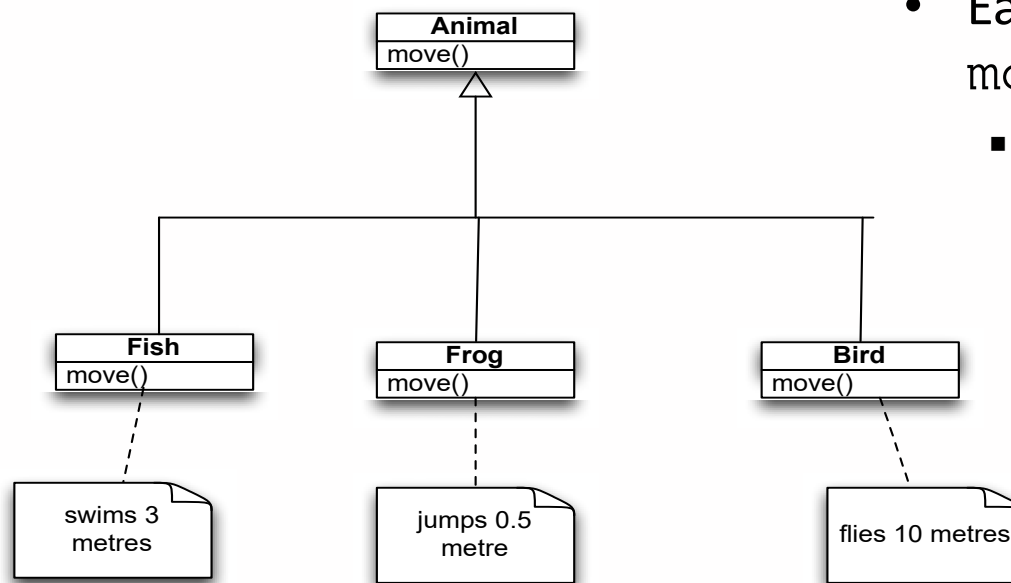
Open a new Project in Blue J, create an abstract class called **Animal** with one abstract method *move*

Write the code for three subclasses: **Fish**, **Frog** and **Bird**



Example

- Open a new Project in Blue J, create an abstract class called Animal with one abstract method *move*
- Create three sub-classes of Animal: Fish, Frog, Bird



- Each inherits and **overrides** the `move()` method
 - A Fish swims, a Frog jump, a Bird Flies



Animal Code

```
public abstract class Animal
{
    public abstract void move(int y);
}
```

```
public class Bird extends Animal
{
    @Override
    public void move(int y)
    {
        System.out.printf("I fly %d metres", y);
    }
}
```

```
public class Fish extends Animal
{
    @Override
    public void move(int y)
    {
        System.out.printf("I swim %d metres", y);
    }
}
```

```
public class Frog extends Animal
{
    @Override
    public void move(int y)
    {
        System.out.printf("I hop %d metres", y);
    }
}
```



Polymorphism Key point

- In general, a variable of type X can point to any object that has an 'is-a' relationship to type X

```
Animal bird1 = new Bird();  
Animal bird2 = new Bird();  
Animal frog1 = new Frog();  
Animal frog2 = new Frog();  
Animal fish1 = new Fish();  
}
```

- A variable of type `Animal` can point to a `Bird`, `Frog` or `Fish` object
- `Bird`, `Frog` or `Fish` objects have an 'is-a' relationship to the `Animal` class



'Is-a' relationship

E.g. a variable of type Animal can point to objects of any type **directly below it** in the class hierarchy



Codepad

Create an array of Animal references of size 6

```
Animal[] animal = new Animal[6];
```

Even though Animal is an abstract class we can still create an array of Animal references



Write the code

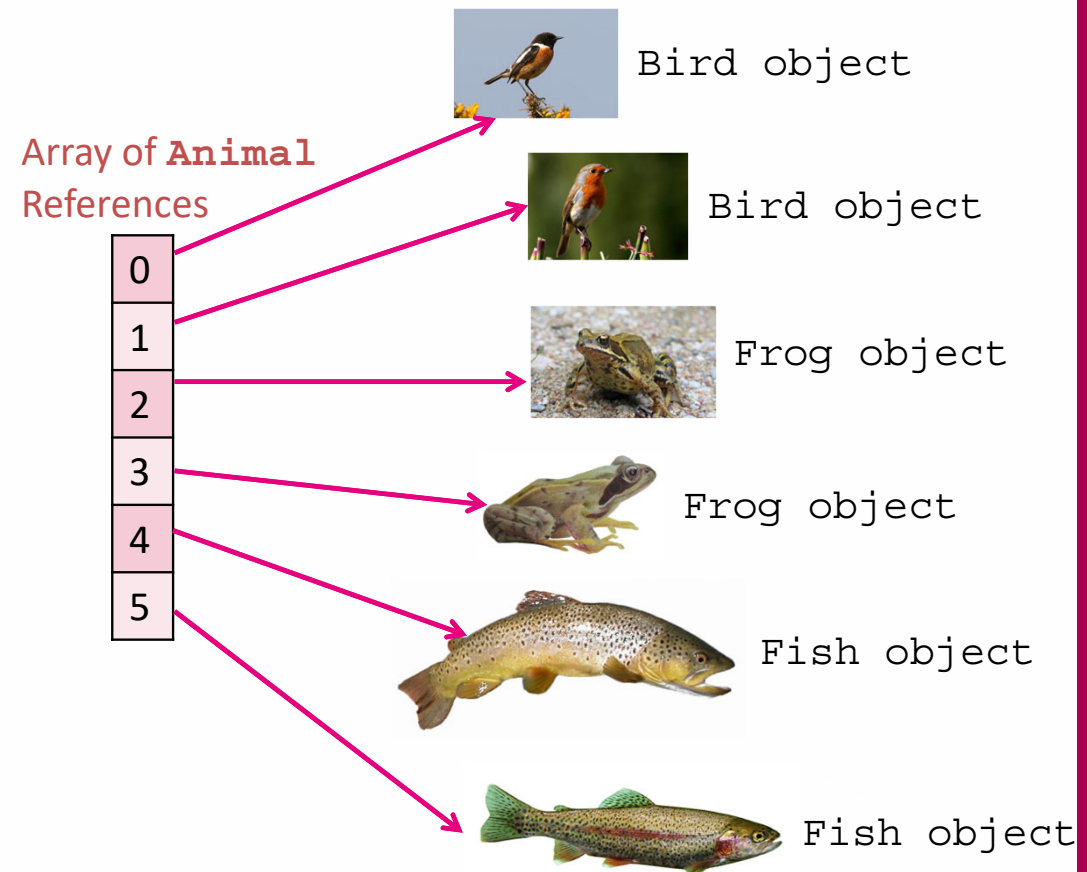
Now write the code to add a reference to a different animal in each array location

E.g. a bird in the first location

A bird in the second location

A Frog in the third location

And so on



For tomorrow, write the code requested in the previous slide in a new Class with a main method.





OLLSCOIL NA GAILLIMHE
UNIVERSITY OF GALWAY

CT2106

Object Oriented Programming



Dr. Frank Glavin
Room 404, IT Building
Frank.Glavin@UniversityofGalway.ie
School of Computer Science

University
ofGalway.ie

Lecture Topic

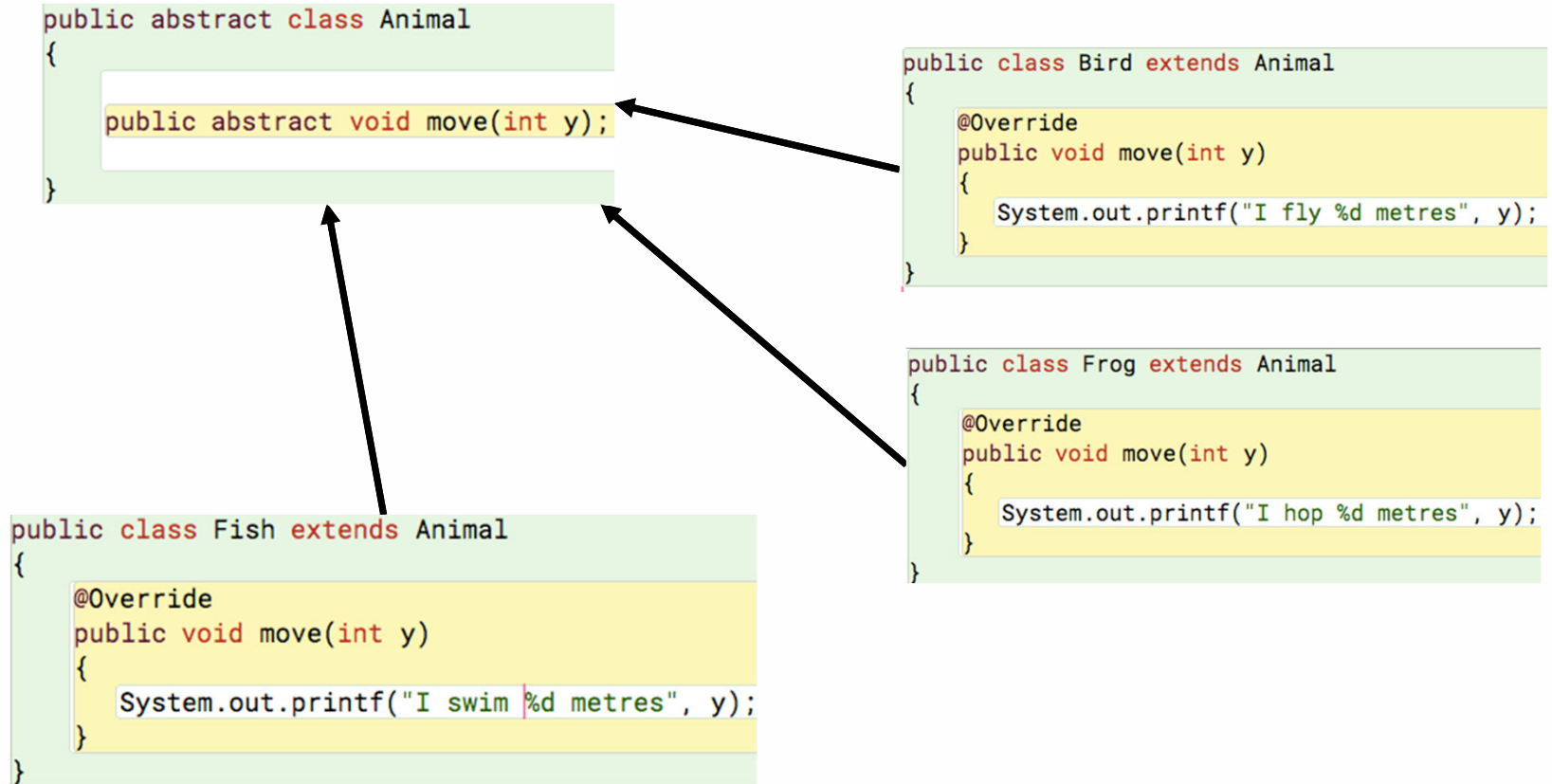
Polymorphism

For examples, see: <https://www.javatpoint.com/runtime-polymorphism-in-java>



OLLSCOIL NA GAILLIMH
UNIVERSITY OF GALWAY

Animal Code



Write the code

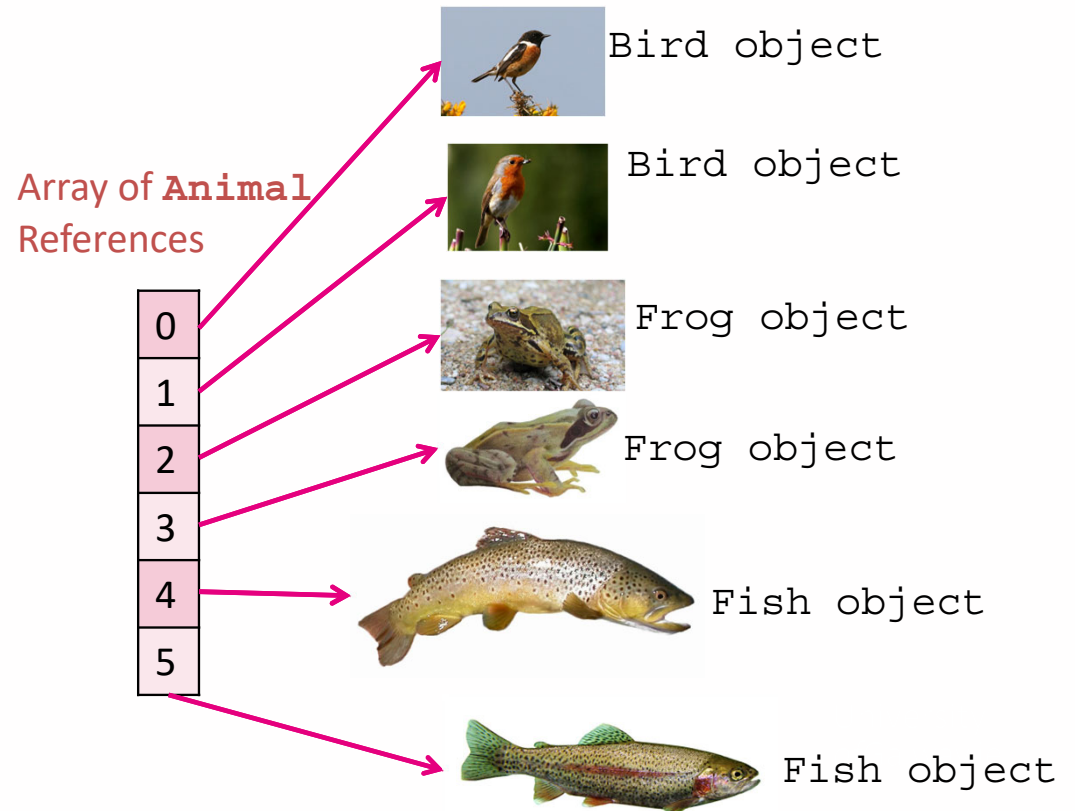
Write the code to add a reference to a different animal in each array location

E.g. a bird in the first location

a bird in the second location

A Frog in the third location

And so on



```
public class AnimalTest  
{
```

```
public static void main(String[] args)
```

```
{  
    Animal[] animals = new Animal[6];  
    animals[0] = new Bird();  
    animals[1] = new Bird();  
    animals[2] = new Frog();  
    animals[3] = new Frog();  
    animals[4] = new Frog();  
    animals[4] = new Fish();  
    animals[5] = new Fish();  
}
```



Bird object



Bird object



Frog object



Frog object



Fish object



Fish object



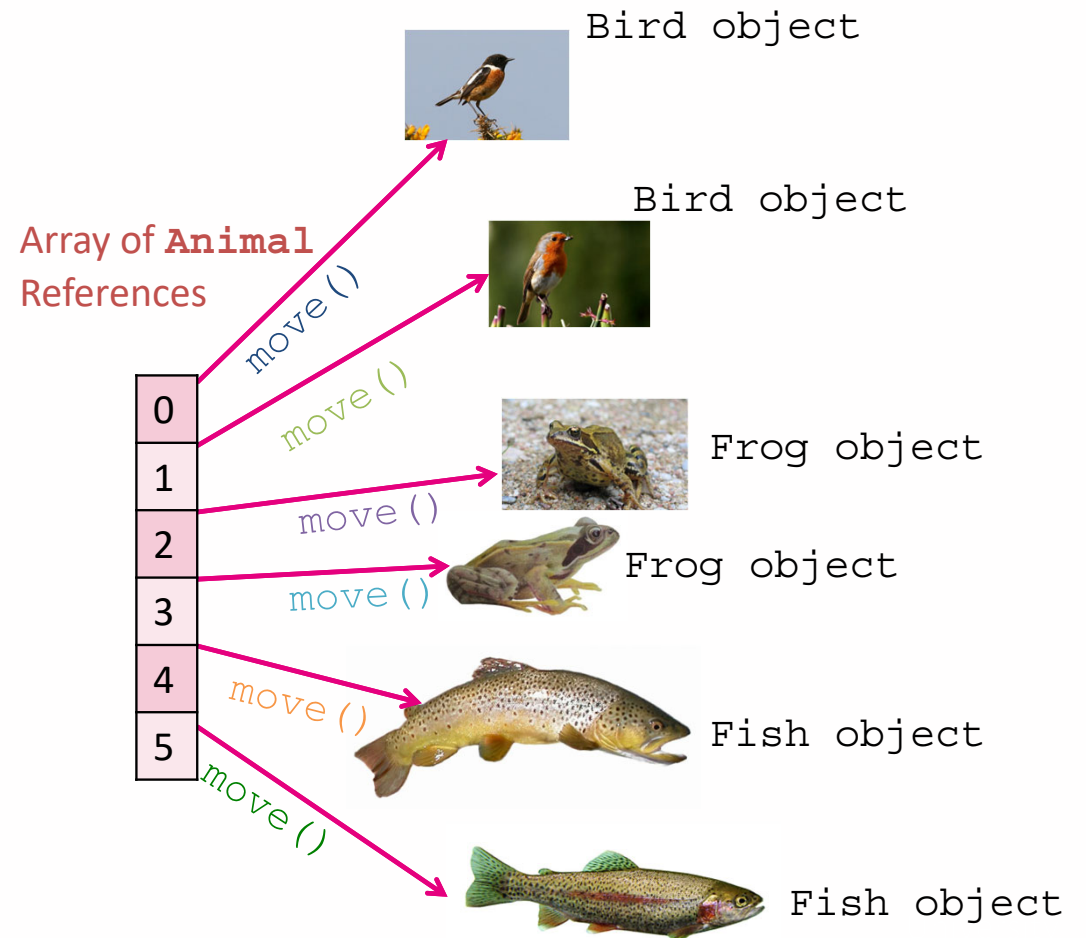
Example

Now write the code

to call the `move()` method
from each
reference in the array
Use a *for* loop

```
for(Animal animal: animals){  
    animal.move(5);  
}
```

Run the code from the main
method



Example

- Note how you haven't explicitly called the move methods of Bird, Frog or Fish
- Just the move method of Animal (which is abstract)

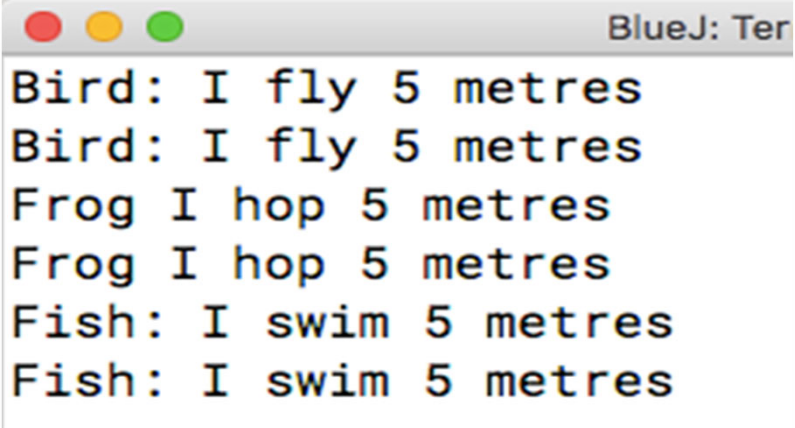
```
for(Animal animal: animals){  
    animal.move(5);  
}
```



Output

Examine the output produced in the terminal

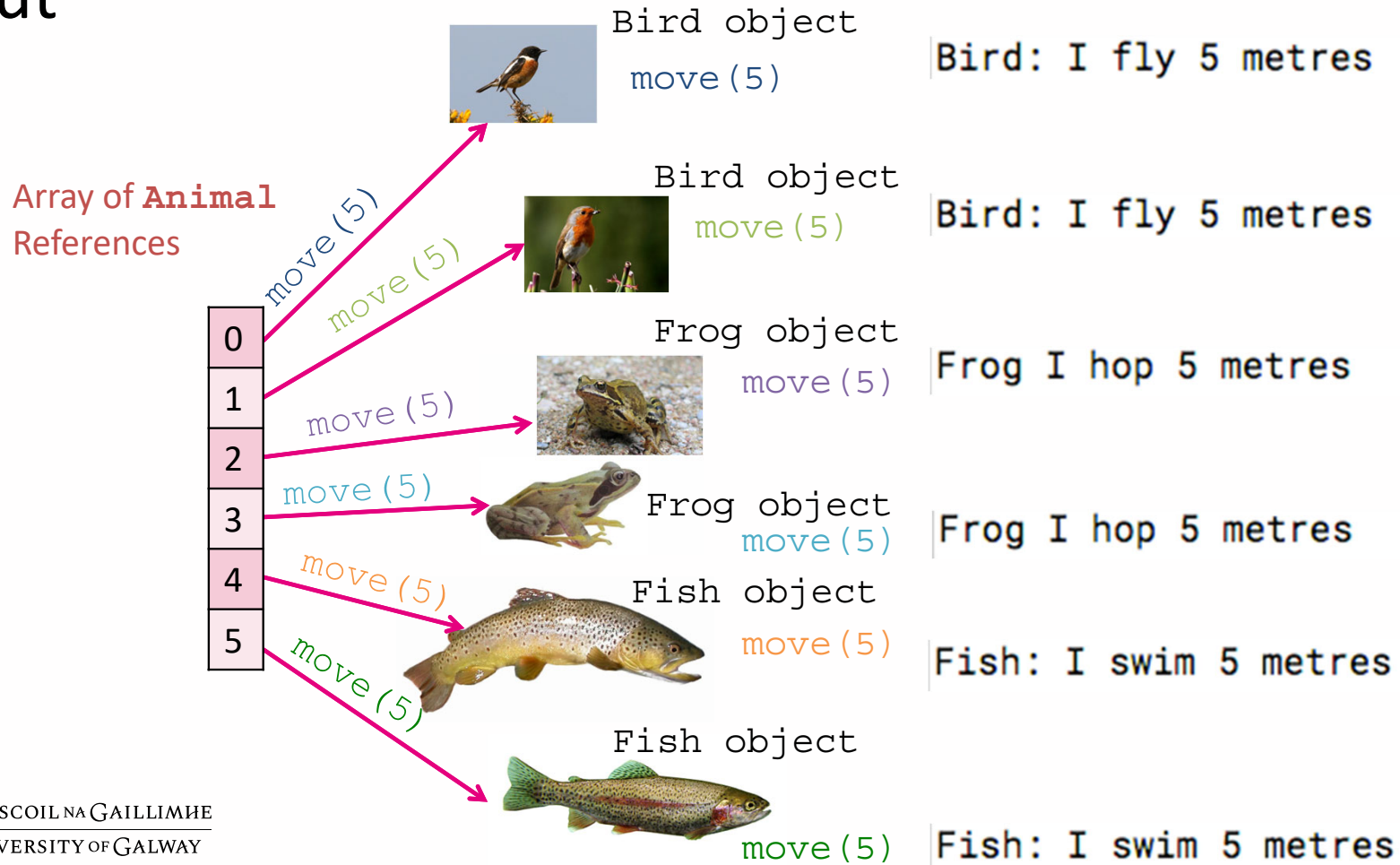
The specific *move* method of each of the referenced animal objects(Bird, Frog, Fish) has been called



```
BlueJ: Ter
Bird: I fly 5 metres
Bird: I fly 5 metres
Frog I hop 5 metres
Frog I hop 5 metres
Fish: I swim 5 metres
Fish: I swim 5 metres
```

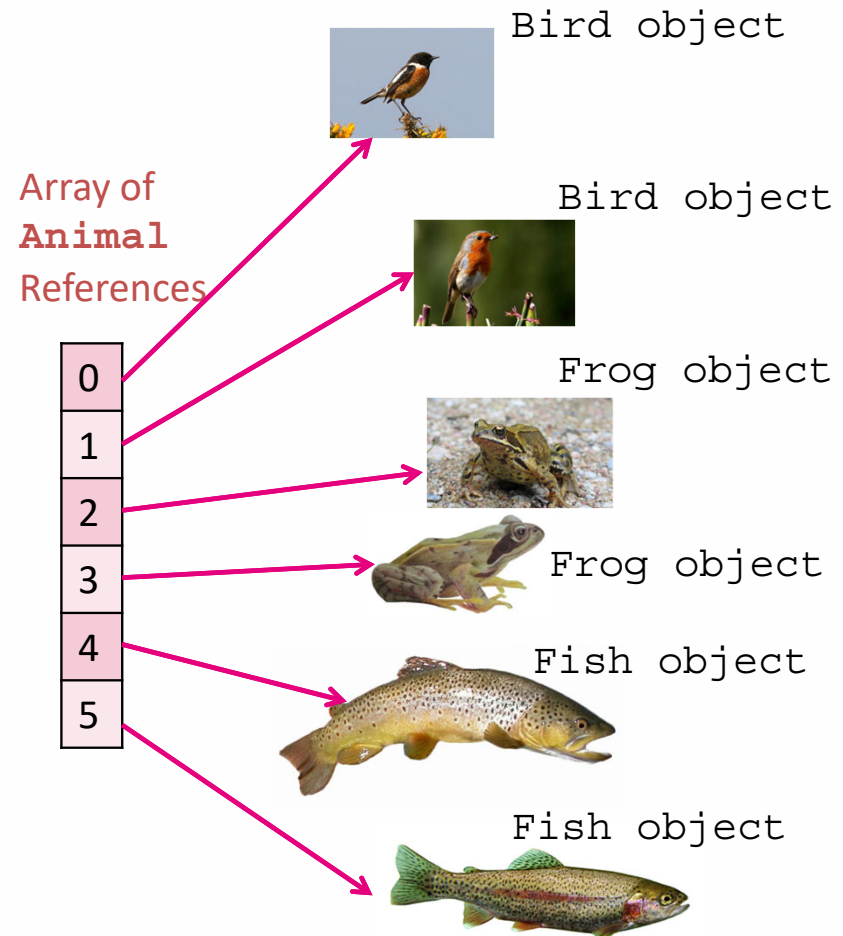


Output



Explanation

- Each element in the array contains a reference variable of type `Animal`
- Each reference points to a `Bird`, `Frog` or `Fish` object
- So when the `move()` method is called from the `Animal` references in the array it is the `move()` method of the respective `Bird`, `Frog`, `Fish` objects that is invoked



Dynamic Dispatch/Late binding

- This an example of what is called **dynamic dispatch** or **late binding**
- The decision as to which method to invoke is decided at program runtime, not compilation time
- If at run time, `animals[0]` points to a `Bird` object, then `animals[0].move()` invokes the `move()` method of the `Bird` object
- If `animals[0]` points to a `Fish` object, then `animals[0].move()` invokes the `move()` method of the `Fish` object

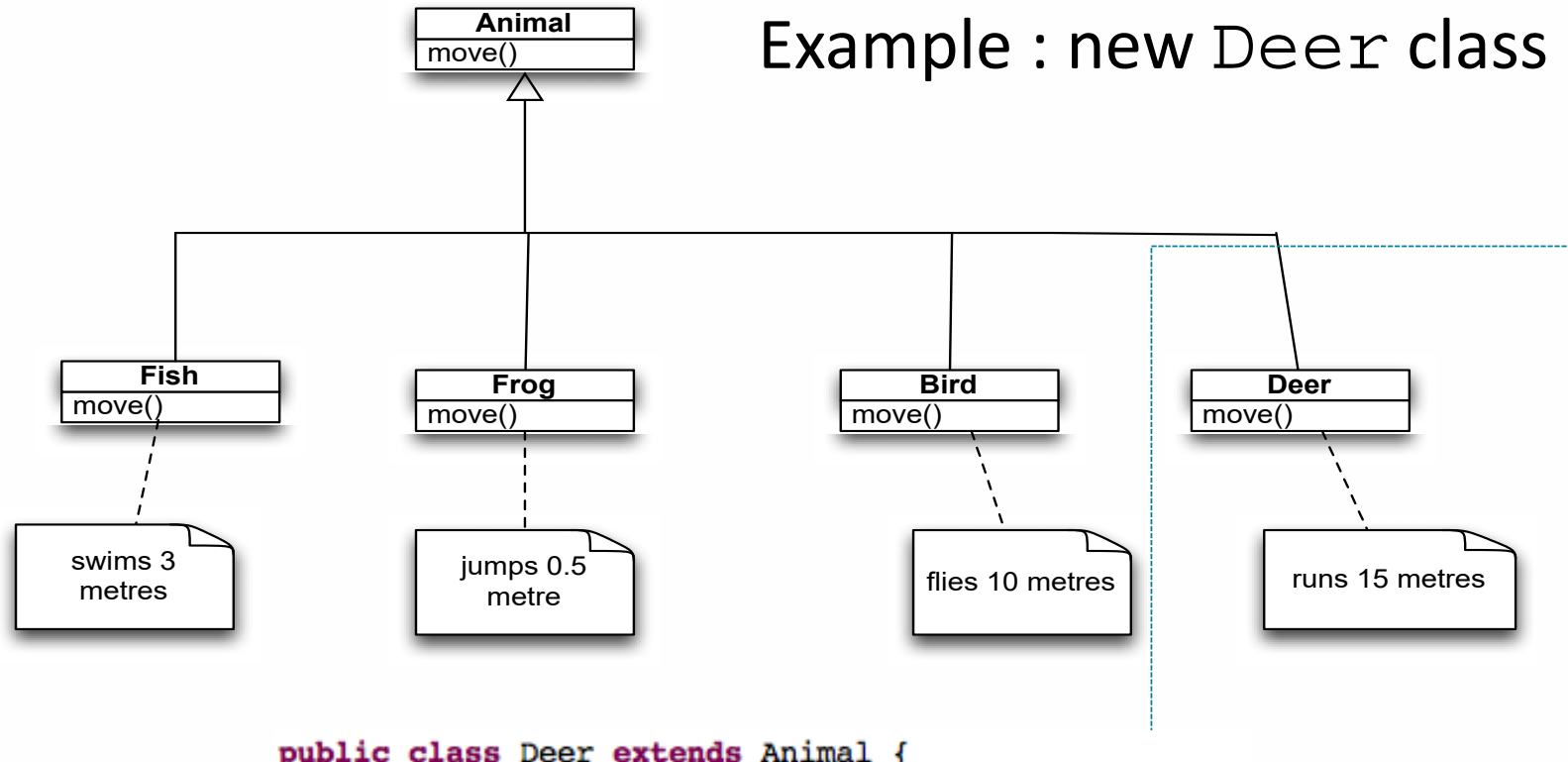


Polymorphism

- We can add new `Animal` types with new `move()` behaviours to the array of `Animal` references
- As long as these are subclasses of `Animal`, their `move()` method will always be called



Example : new Deer class



```
public class Deer extends Animal {

    @Override
    public void move(){
        // TODO code for flying 10 metres
        System.out.println("Deer: I run 15 metres");
    }
}
```



Create a deer object

- Place a reference to a Deer object in the array and run the program again.

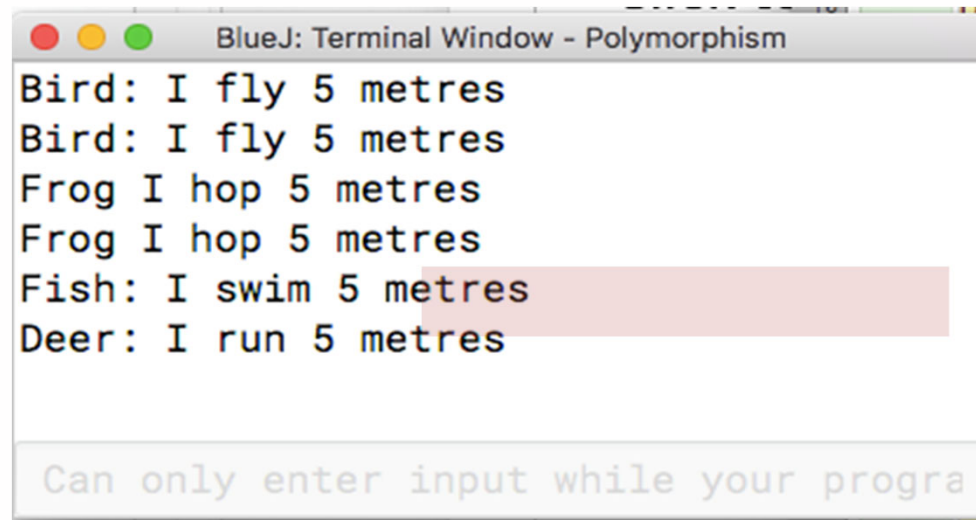
```
Animal[] animals = new Animal[6];
animals[0] = new Bird();
animals[1] = new Bird();
animals[2] = new Frog();
animals[3] = new Frog();
animals[4] = new Frog();
animals[4] = new Fish();
animals[5] = new Fish();
animals[5] = new Deer(); // this replaces the previous value

for(Animal animal: animals){
    animal.move(5);
}
```



Output

- Key message we can change the behaviour of a program **without changing its code**
- E.g. this piece of code remains the same



```
BlueJ: Terminal Window - Polymorphism
Bird: I fly 5 metres
Bird: I fly 5 metres
Frog I hop 5 metres
Frog I hop 5 metres
Fish: I swim 5 metres
Deer: I run 5 metres

Can only enter input while your progra
```

```
for(Animal animal: animals){
    animal.move(5);
}
```



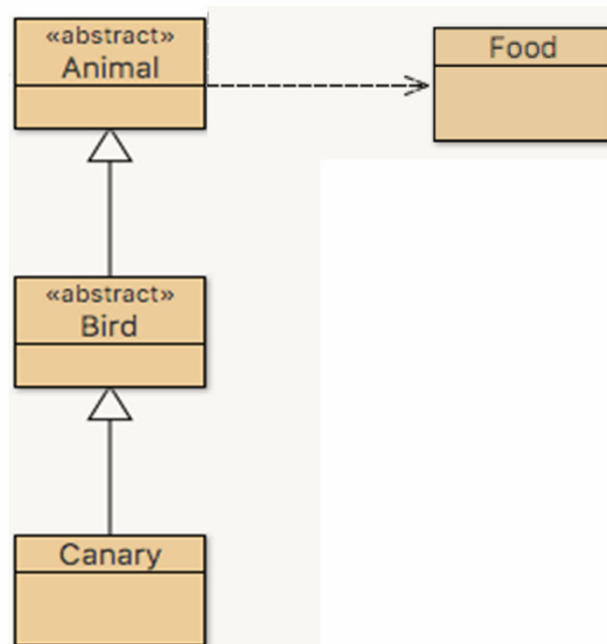
Implications

- With polymorphism, we can design and implement systems that are *easily extensible*
- New classes with new behaviours can be added with little or no modification to the general portions of the program



Let's look at applying these ideas

Open the code we first looked at yesterday



Instructions

Food:

Make Food an abstract class

Give it two abstract methods *getCalories* and *getFat* with a return type *int*

Animal: make *eat* method abstract

- Create an abstract subclass of Food called **Vegetable**
- Create a concrete subclass of Vegetable called **Seed**
- Seed has two fields *calories* and *fat*
- Canary must implement a concrete version of the *eat* method
- Canary's *eat* method checks if Food object is an *instanceof* Seed; if it is, the Canary calls Food's *getCalories* method and moves the distance returns. She also calls the *sing* method.



Lecture wrap up

- We looked at polymorphism – the facility by which an object can be referenced by a variable of its Superclass
- This allows us to create code that is easily extensible
- We saw that we can create variables of abstract types (classes)





OLLSCOIL NA GAILLIMHE
UNIVERSITY OF GALWAY

CT2106

Object Oriented Programming



Dr. Frank Glavin
Room 404, IT Building
Frank.Glavin@UniversityofGalway.ie
School of Computer Science

University
ofGalway.ie

Instructions from last week

Food:

Make Food an abstract class

Give it two abstract methods *getCalories* and *getFat* with a return type *int*

Animal: make *eat* method abstract

- Create an abstract subclass of Food called **Vegetable**
- Create a concrete subclass of Vegetable called **Seed**
- Seed has two fields *calories* and *fat*
- Canary must implement a concrete version of the *eat* method
- Canary's *eat* method checks if Food object is an *instanceof* Seed; if it is, the Canary calls Food's *getCalories* method and moves the distance returns. She also calls the *sing* method.



Slight revision to these instructions

- We'll drop the *getFat* method from Food – as I don't plan to use it
- Canary's *eat* method should do the following:
 - Check if the Food object is null
 - Checks if Food object is an *instance of* Seed;
 - if it is a Seed, the canary calls Food's *extractEnergy* method ~~and moves the distance~~ ~~returns~~ and adds the value returned to its own energy level
 - It also calls the sing method (because it is now well fed)



This lecture

- We'll look at some modelling issues
- We'll introduce the background for the next topic: **interfaces**
- To introduce this topic we'll model a **food chain**



Food Chain

Download the zip file provided in the Week 8 folder

Create a new Project in BlueJ

In the Workbench menu, select

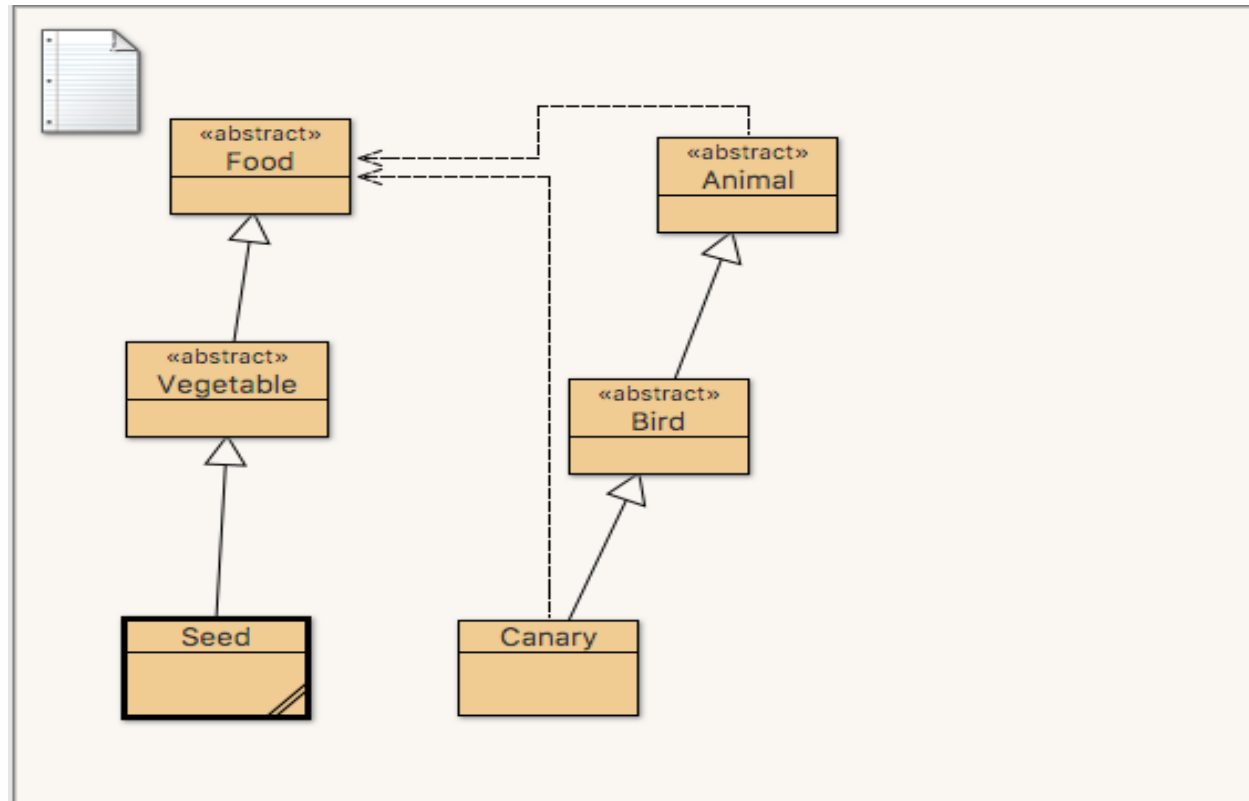
Project -> Open Zip/Jar

Then compile the Project

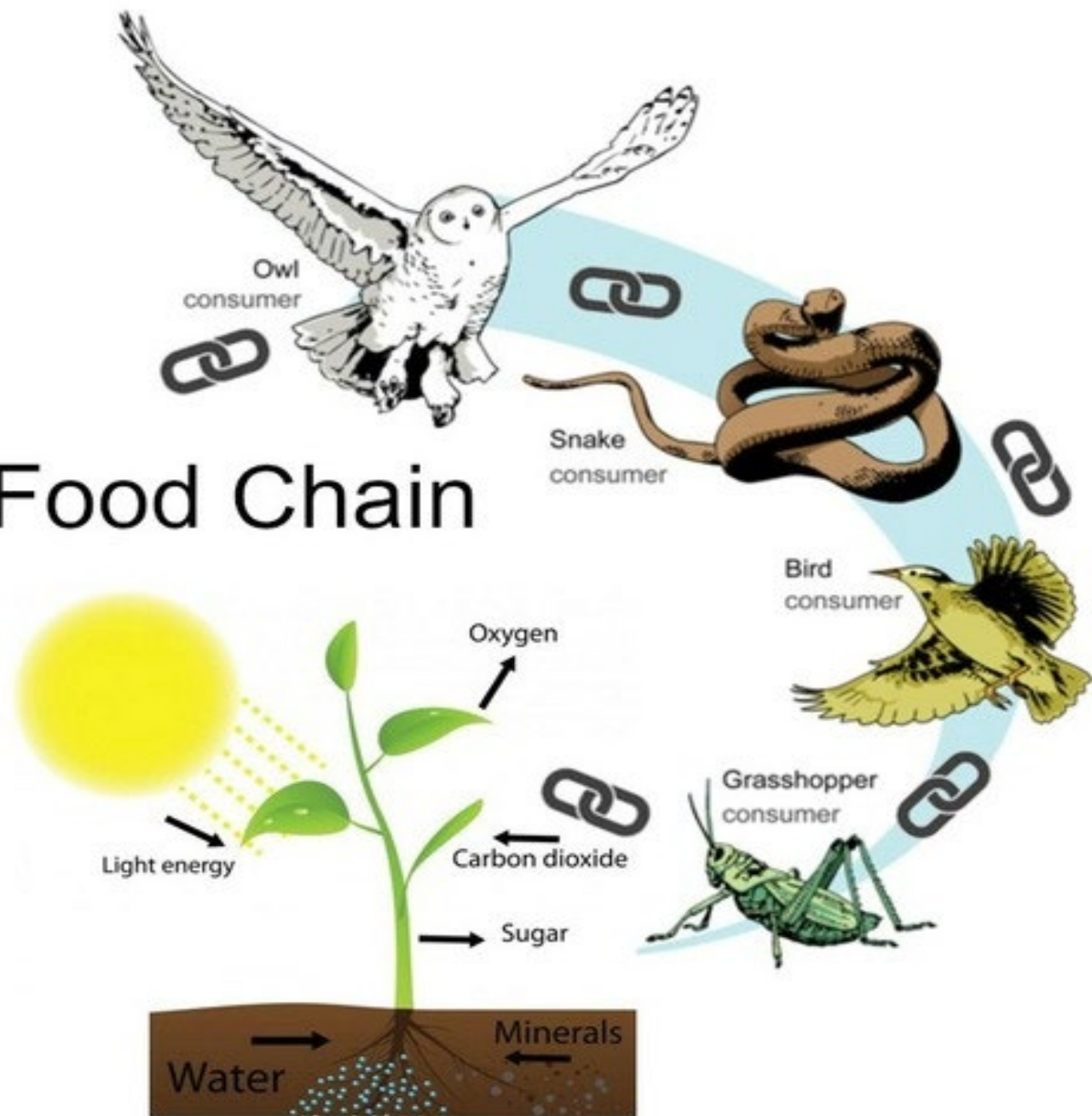


Blue J workbench

Rearrange the class icons to give you something like



Food Chain



Our Food Chain



Seeds



Canaries



Cats

- Canaries eat Seed
- Cats eat Canaries
- Energy passes from Seeds to the Canary to the Cat



Canaries eat Seed

- Animal class has an abstract eat method
- Canary has to *override* the eat method it has inherited from Animal
- We now have to write the specific code to allow Canaries eat Seed

```
@Override  
public void eat(Food food){  
    // TODO  
}
```

Note how the eat method takes as input a Food reference



Canary's eat method

- Canary's *eat* method should do the following:
 1. Check if the Food object is null
 2. Checks if Food object is an *instance of* Seed;
 3. If it is a Seed, the canary calls the *extractEnergy* method and *adds* the value returned to its own energy level
 4. It also calls the *sing* method (because it is now well fed)
- I would also suggest that this method is modified to return a boolean depending on whether the Food is edible (e.g it is a Seed or not)



First: Animal energy

As an Animal object gets energy from the Food objects it can consume, it needs a numeric field *energy* to hold this value

This field can then be inherited by all Animal objects, including Canary

```
public abstract class Animal
{
    // instance variables - replace the example
    boolean hasSkin;
    boolean breathes;
    String colour;
    int energy;
```



getEnergy

You will also need an accessor (getter) method for the new energy field in Animal

```
/**
 * getter method for energy field
 * All subclasses inherit this method
 */
public int getEnergy(){
    return energy;
}
```

Please remember Getter/Setter methods are not optional.
You must use them to access the fields of an object



extractEnergy

An abstract method defined in the Food class

It must be implemented in one of the subclasses of Food

We implement it in the Seed Class. **Implement this method, as described**

```
/**
 * returns the current value for Calories
 * and then sets the calory value to zero
 * i.e. the energy has been extracted from Seed
 */
@Override
public int extractEnergy(){
    //TODO
    return 0;
}
```



All Food has calories

I originally declared the *calories* field in the Seed class

But *all* Food has calories

Therefore, we should remove the *calories* declaration in Seed and move it to the Food class

```
public abstract class Food
{
    // instance variables - replace t
    int calories;
```

It can be then inherited by all sub-classes of Food, including Seed



Implement Canary's eat method

Canary's *eat* method should do the following:

1. Check if the Food object is null
2. Checks if Food object is an *instanceof* Seed;
3. If it is a Seed, the canary calls the *extractEnergy* method and *adds* the value returned to its own energy level
4. It also calls the sing method (because it is now well fed)

I would also suggest that this method is modified to return a boolean depending on whether the Food is edible (e.g it is a Seed or not)



Test first part of the food chain



Seeds



Canaries

- Each seed has 10 calories
- If a Canary eats 3 seeds, its energy level should be 30



In Code Pad

Or in a main method, type the following

```
Seed millet = new Seed();  
Seed sunflower = new Seed();  
Seed hayseed = new Seed();  
Canary bluey = new Canary("Bluey");  
bluey.eat(millet);  
bluey.eat(sunflower);  
bluey.eat(hayseed);  
System.out.println(bluey.getEnergy());
```

This should print out the value 30



Part 2 of our food chain



Seeds



Canaries



Cats

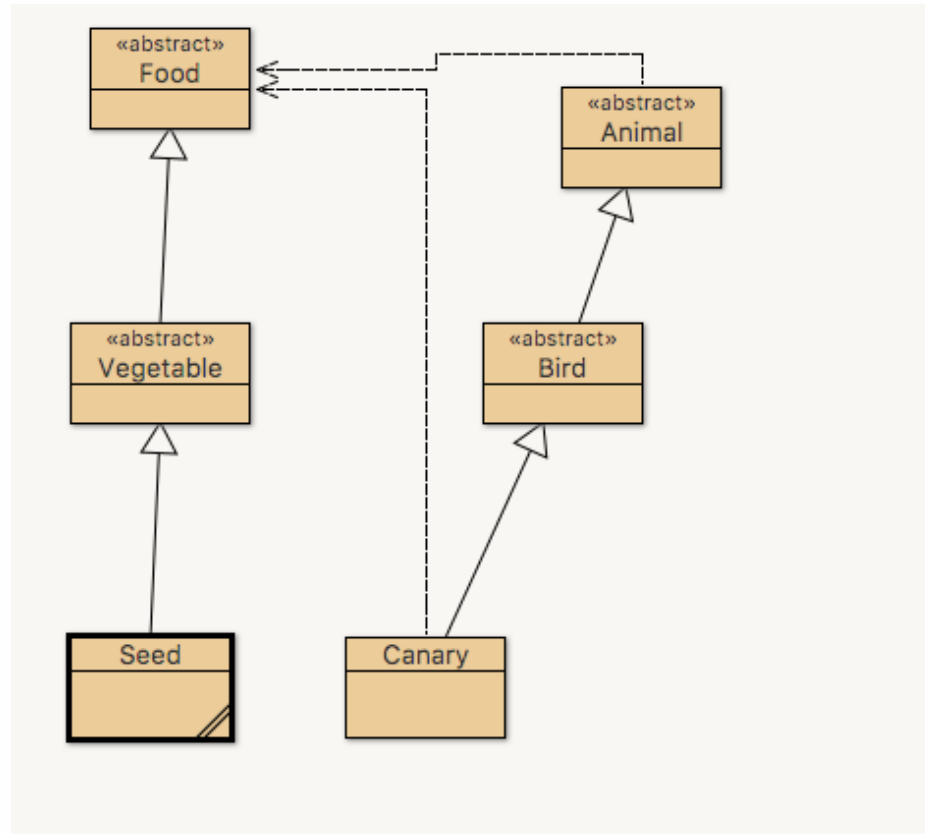
- Cats eat Canaries
- Energy passes from the canary to the Cat



Part 2

Currently the class structure looks like this
You are now going to add two more classes

- Feline (abstract)
- Cat (concrete)



Feline class (abstract)

Extends Animal

Fields

hasFur

Overrides

move() method

Cat class (concrete)

Extends Feline

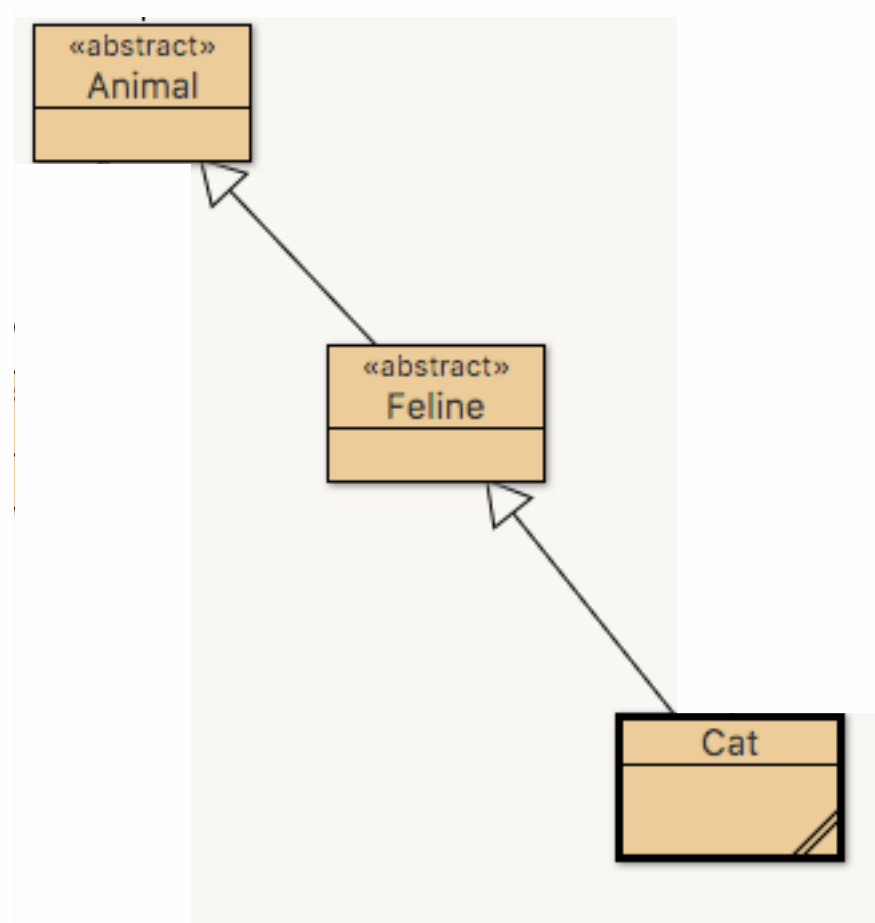
Fields

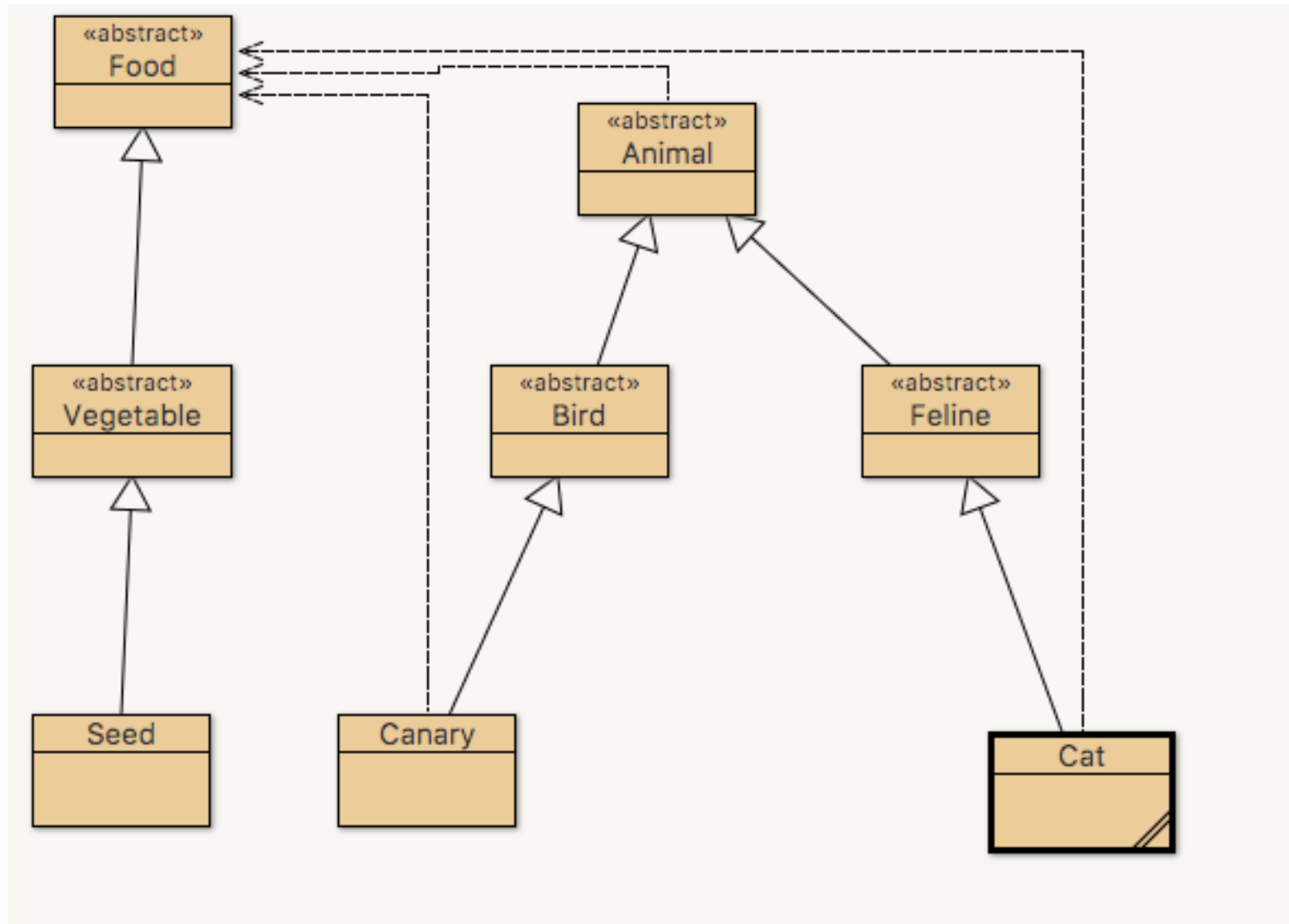
name

Overrides

colour field (colour=black)

eat (Food) method







OLLSCOIL NA GAILLIMHE
UNIVERSITY OF GALWAY

CT2106

Object Oriented Programming



Dr. Frank Glavin
Room 404, IT Building
Frank.Glavin@UniversityofGalway.ie
School of Computer Science

University
ofGalway.ie

Our Food Chain



Seeds



Canaries



Cats

- Canaries eat Seed
- Cats eat Canaries
- Energy passes from Seeds to the Canary to the Cat



Implement Canary's eat method

Canary's *eat* method should do the following:

1. Check if the Food object is null
2. Checks if Food object is an *instanceof* Seed;
3. If it is a Seed, the canary calls the *extractEnergy* method and *adds* the value returned to its own energy level
4. It also calls the sing method (because it is now well fed)

I would also suggest that this method is modified to return a boolean depending on whether the Food is edible (e.g it is a Seed or not)



Eat method

```
public abstract boolean eat(Food food);
```

“The eat method in Animal should be changed to return a boolean value.”

“In Canary's case, the eat method should return *true* if the food variable is an instance of Seed. Otherwise, the method should return **false**.”

```
@Override
public boolean eat(Food food){
    if(food ==null){ // if the reference points to null
        return false; // immediately return. Method execution goes
    }

    if(food instanceof Seed){ // is food pointing to a Seed object
        Seed seed = (Seed) food; // cast reference to a Seed type
        energy+=seed.extractEnergy(); // extract the Seeds energy
        sing(); // sing
        return true; // return. Method execution goes no further
    }else{
        System.out.println("I cannot eat this type of food");
    }
    return false;
}
```



Adding Feline and Cat classes

Feline class (abstract)

Extends Animal

Fields

hasFur

Overrides

move() method

Cat class (concrete)

Extends Feline

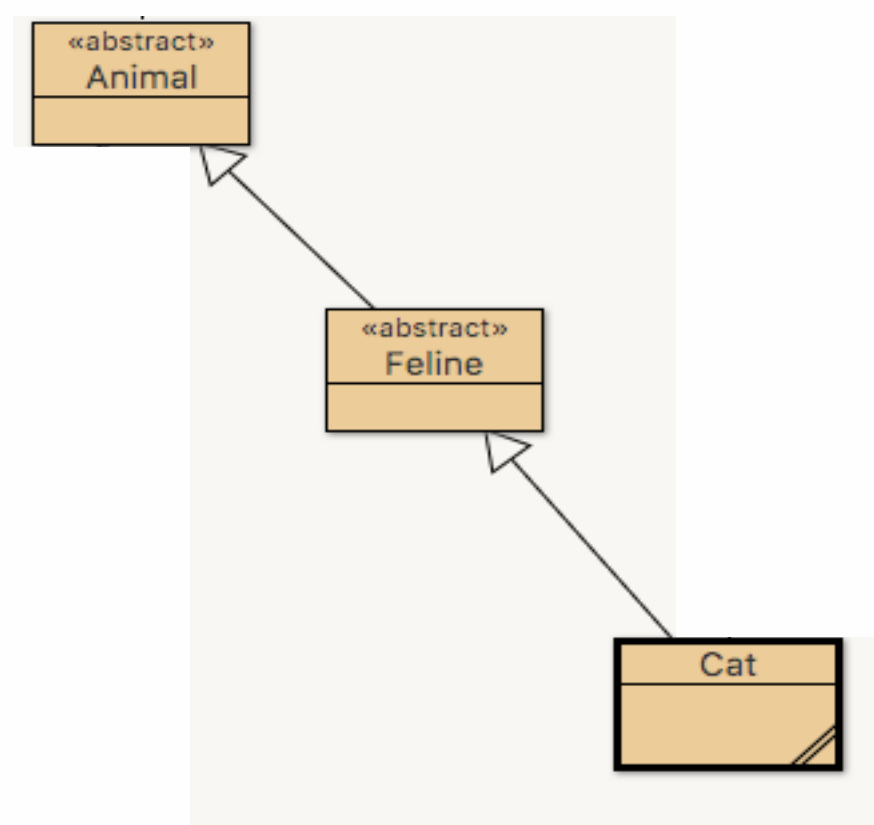
Fields

name

Overrides

colour field (colour=black)

eat (Food) method



Feline class

```
public abstract class Feline extends Animal
{
    boolean hasFur = true;

    @Override
    public void move(int distance)
    {
        System.out.printf("I am a Feline and I leap %d metres, \n", distance);
    }

    public boolean hasFur(){
        return hasFur;
    }
}
```

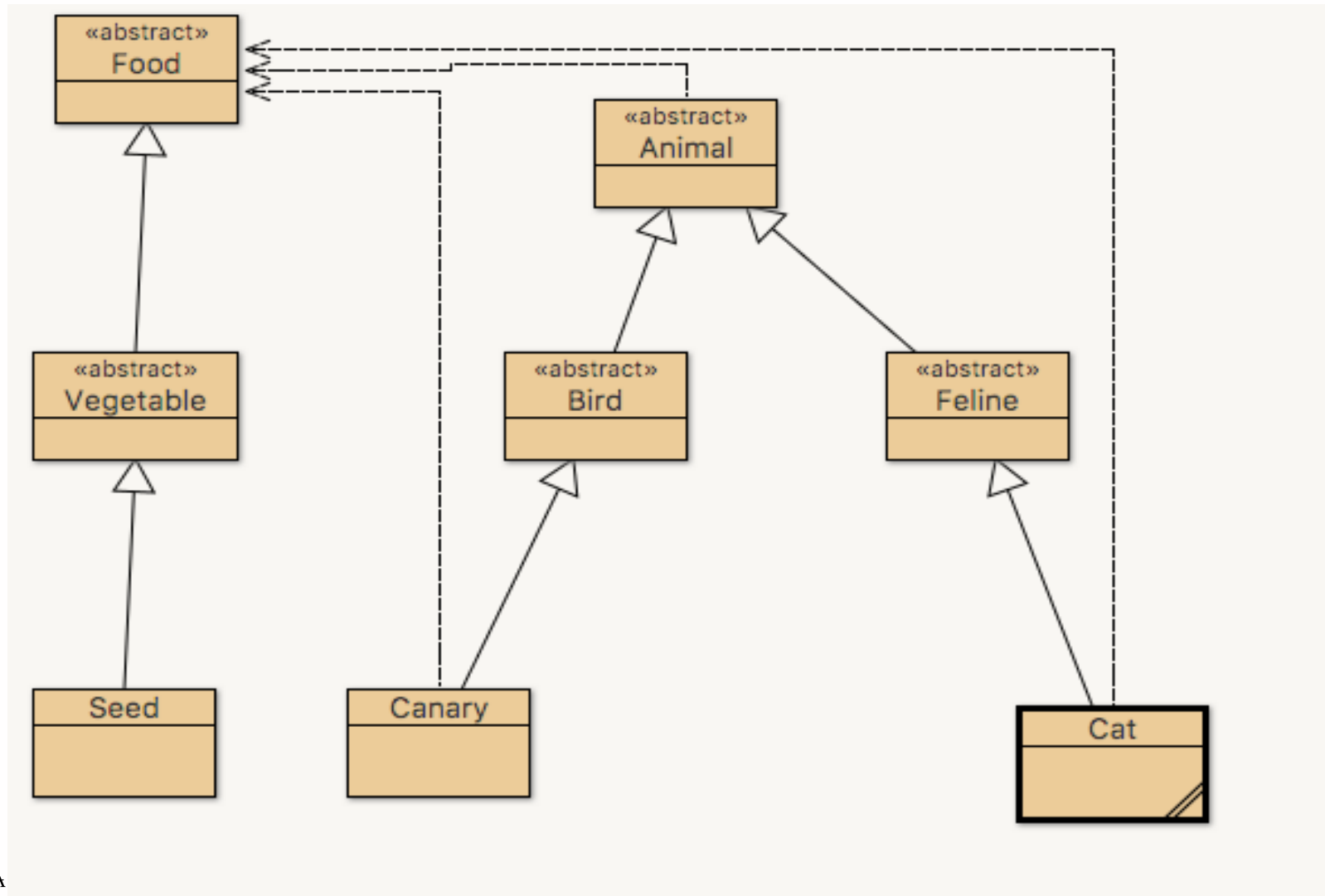


Cat class

```
public class Cat extends Feline
{
    String name;
    /**
     * Constructor for objects of class Cat
     */
    public Cat(String name)
    {
        super();
        colour = "black"; // override default colour from Animal
        this.name = name;
    }

    /**
     * eat method
     * @param Food food : Cats eat Canaries
     * so the method has to make sure that food points to
     * a Canary object
     */
    @Override
    public boolean eat(Food food)
    {
        //TODO
        return false; // default return value
    }
}
```





eat method of Cat

For this to work, a Canary **must** be a subclass of Food, just as Seed is
However, this is not the case.
Canary is a subclass of Animal

```
/**
 * eat method for a Cat
 * In this programme Cats eat Canary objects only
 * @param Food
 */
@Override
public void eat(Food food)
{
    // TODO
}
```



A Canary is not a Food type

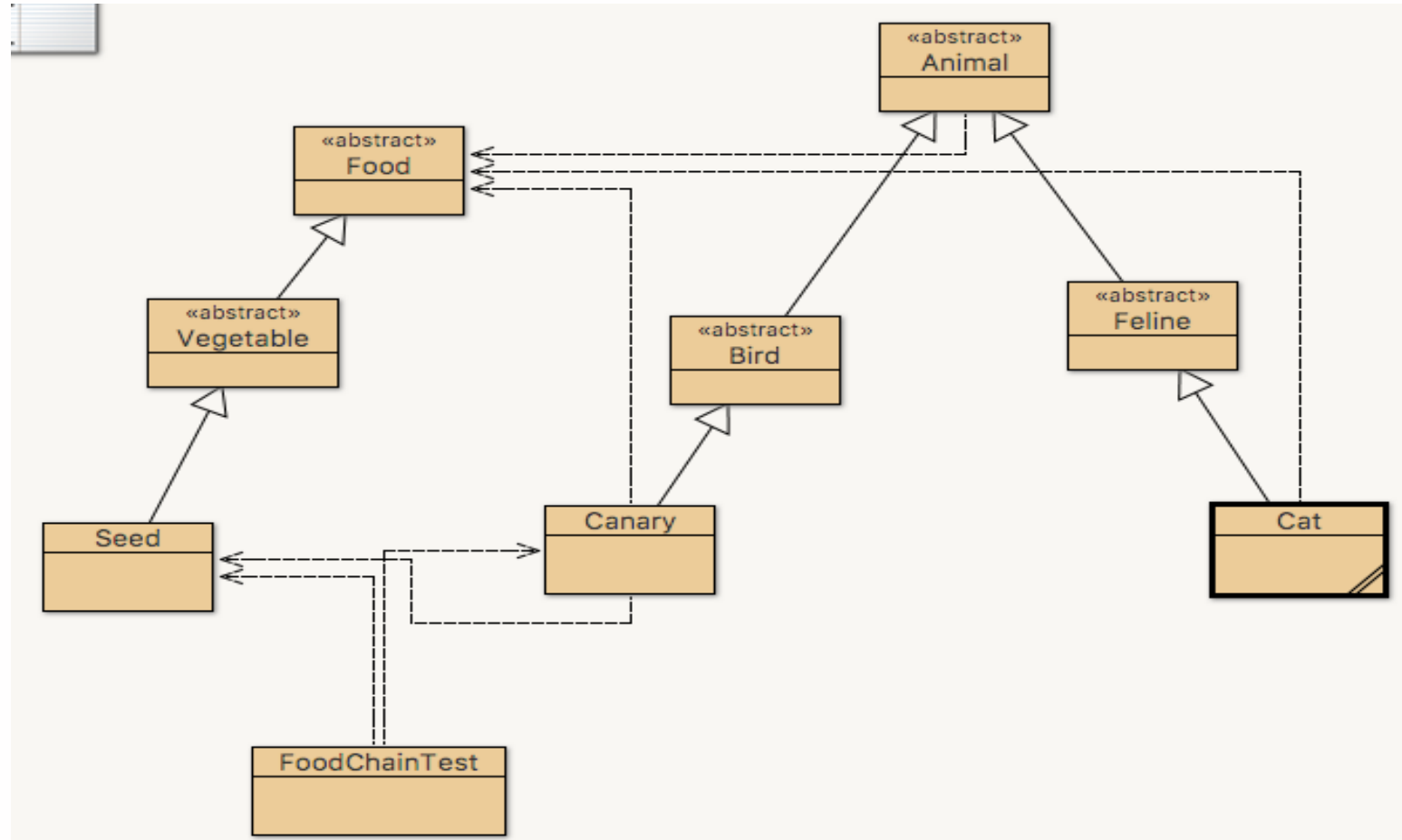
Furthermore, there is no way to cast a Canary object to Food
E.g. Try the following in code pad

```
Food food = new Cat("Felix");  
    Error: incompatible types: Cat cannot be converted to Food  
Cat cat = new Cat("Felix");  
Food food = (Food)cat;  
    Error: incompatible types: Cat cannot be converted to Food
```

For polymorphism to occur, Cat would have to be a subclass of Food



Arrange your classes to look like this



Now open the *eat* method of Cat

Copy and paste the body of the eat method in Canary into this method. Modify

Remember a Cat can only eat a Canary

A Cat doesn't sing

```
/**
 * eat method for a Cat
 * In this programme Cats eat Canary objects only
 * @param Food
 */
@Override
public void eat(Food food)
{
    // TODO
}
```



What problems did you experience?

```
/**
 * eat method
 * @param Food food : Cats eat Canaries
 * so the method has to make sure that food points to
 * a Canary object
 */
public boolean eat(Food food)
{
    if(food ==null){ // if the reference points to null
        return false; // immediately return. Method execution goes no further
    }

    if(food instanceof Canary){ // is food pointing to a Canary object?
        Canary canary = (Canary) food; // cast reference to a Canary type
        energy+=canary.extractEnergy(); // extract the Canary's energy
        //sing(); // cats don't sing
        return true; // return. Method execution goes no further
    }else{
        System.out.println("I cannot eat this type of food");
    }
    return false;
}
```



Incompatible Types

```
/**
 * eat method
 * @param Food food : Cats eat Canaries
 * so the method has to make sure that food points to
 * a Canary object
 */
public boolean eat(Food food)
{
    if(food ==null){ // if the reference points to null
        return false; // immediately return. Method execution goes no further
    }

    if(food instanceof Canary){ // is food pointing to a Canary object?
        // incompatible types: Food cannot be converted to Canary type
        energy=canary.extractEnergy(); // extract the canary's energy
        //sing(); // cats don't sing
        return true; // return. Method execution goes no further
    }else{
        System.out.println("I cannot eat this type of food");
    }
    return false;
}
```



eat method of Cat

Big Problem! Food cannot be converted to Canary

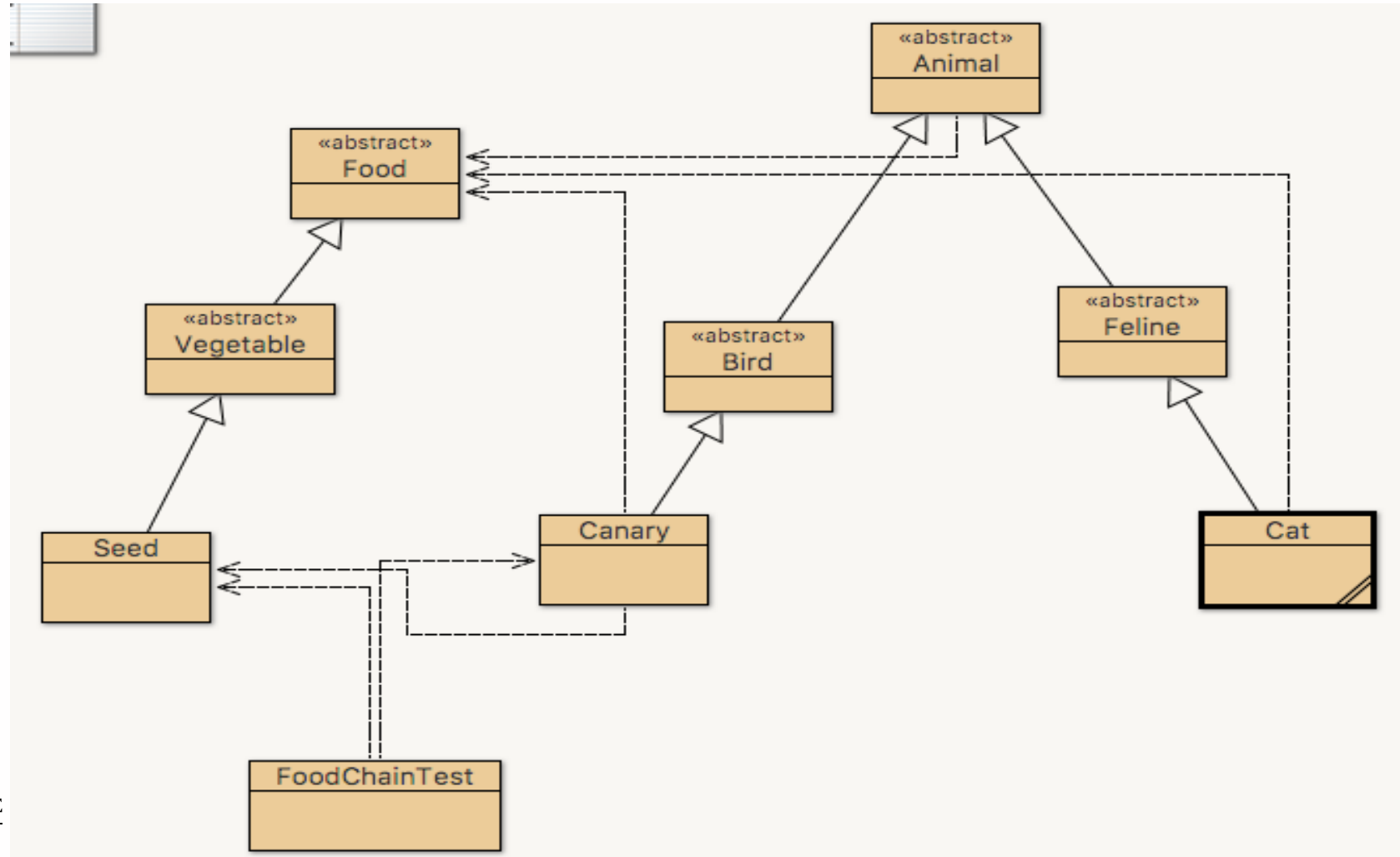
However, the *eat* method only takes a Food reference as an input

In order to convert the Food reference to a Canary reference, Canary **must** be a subclass of Food, just as Seed was

But Canary is a subclass of Animal

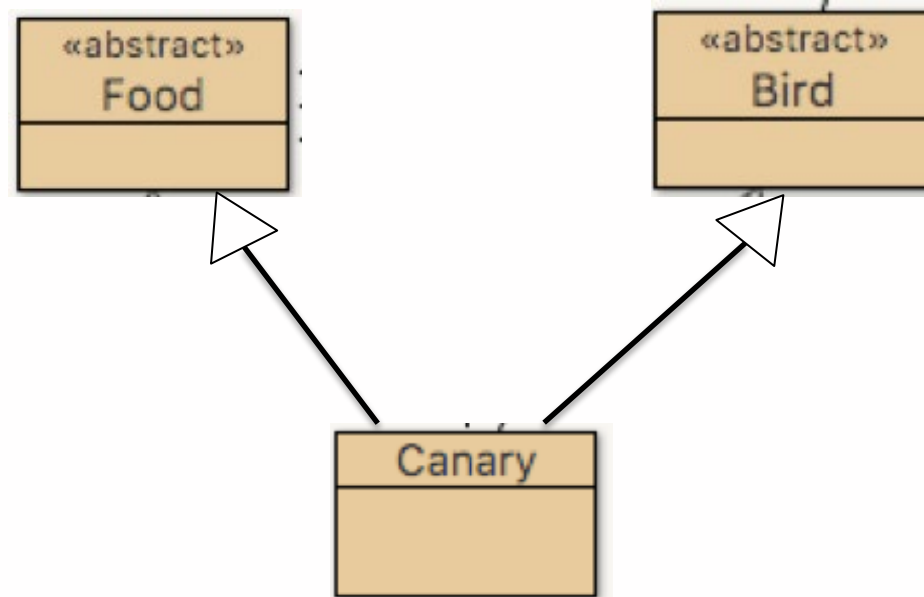


A Canary is not a Food Type



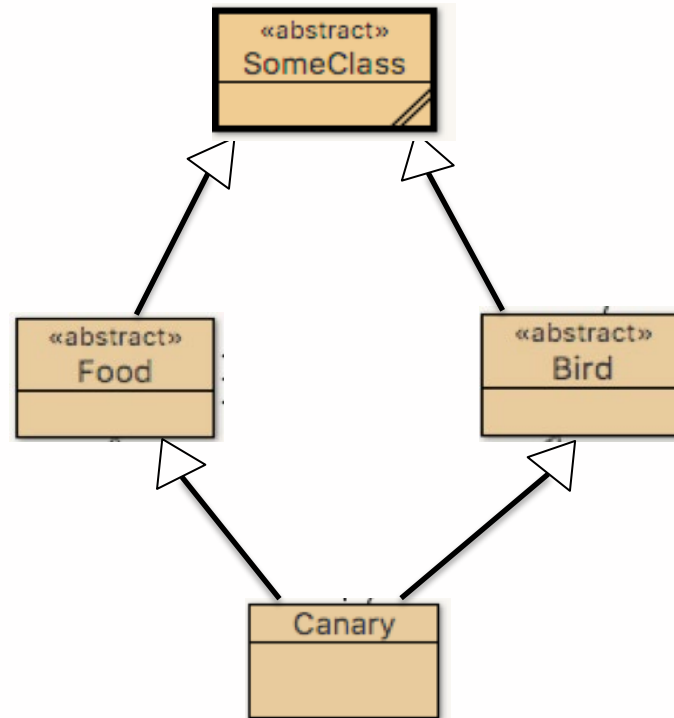
Multiple Inheritance

This problem could be solved using **multiple inheritance** – where a class can have multiple simultaneous superclasses



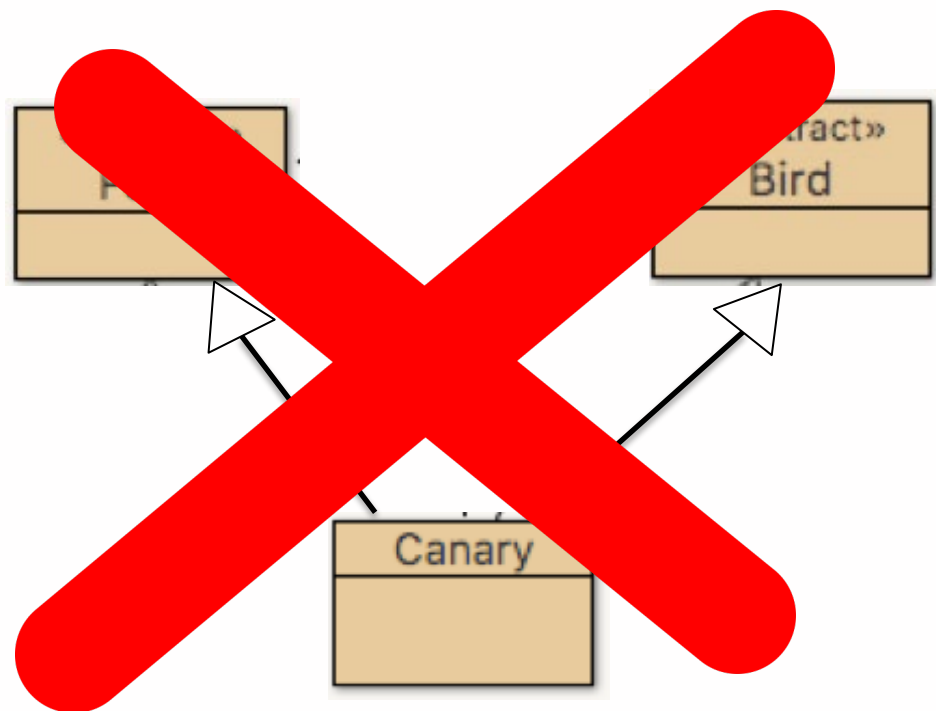
Multiple Inheritance

However, in OOP multiple inheritance has led to **major problems** due to conflicting field and method implementations inherited from superclasses



Multiple Inheritance

Java does not support multiple inheritance



Interface

Java uses a structure called an **interface** to achieve a form of multiple inheritance

An interface is **like a class** – but it is really more like an outline of what methods a class should have

Just like a class an interface can be used **as a type**

Interface names often end in – **able** - simply by convention



Interface example

Compare and Contrast with a class definition

```
public interface Eatable
{
    public int getCalories();
    public int extractEnergy();
}
```



Interface example

Note interface not class

```
public interface Eatable
{
    public int getCalories();
    public int extractEnergy();
}
```

- Note method definitions have no body



Eatable interface

What does it mean?

1. Any class that implements Eatable can be treated as an Eatable type (Polymorphism)
2. Any class that implements Eatable must provide **concrete implementations** of its method



Implementing an interface

While a class can only extend one superclass (direct inheritance)
It can implement **multiple** interfaces



Food as an interface

What does it mean?

1. Any class that **implements** Food can be treated as a Food type (Polymorphism)
2. Any class that implements Food must provide **concrete implementations** of its method



Implementing an interface

A class can only extend one superclass (direct inheritance)

A class can implement **multiple** interfaces

the following class declaration is valid:

```
public class Canary extends Bird implements Food, Comparable{  
...  
}
```

“A Canary is a subclass of Bird and implements the interfaces Food and Comparable”



Solving the Cat's eating problem

We are going to make the Food class into an interface

Any object that is edible (in our domain) will be required to implement the Food interface.



Step 1:

- Change Food to be an interface

```
public interface Food
{
    public int getCalories();
    public int extractEnergy();
}
```

- This also will require Vegetable to **implement** the Food interface
- Seed will need to have its own version of the *calories* field



Step 2

We want Canary to be considered a type of Food

Therefore, Canary should implement the Food Interface

```
public class Canary extends Bird implements Food  
{
```

Canary will be required to implement the Food interface's two methods

getCalories

extractFood



Step 2

Canary should implement Food

```
public class Canary extends Bird implements Food  
{
```

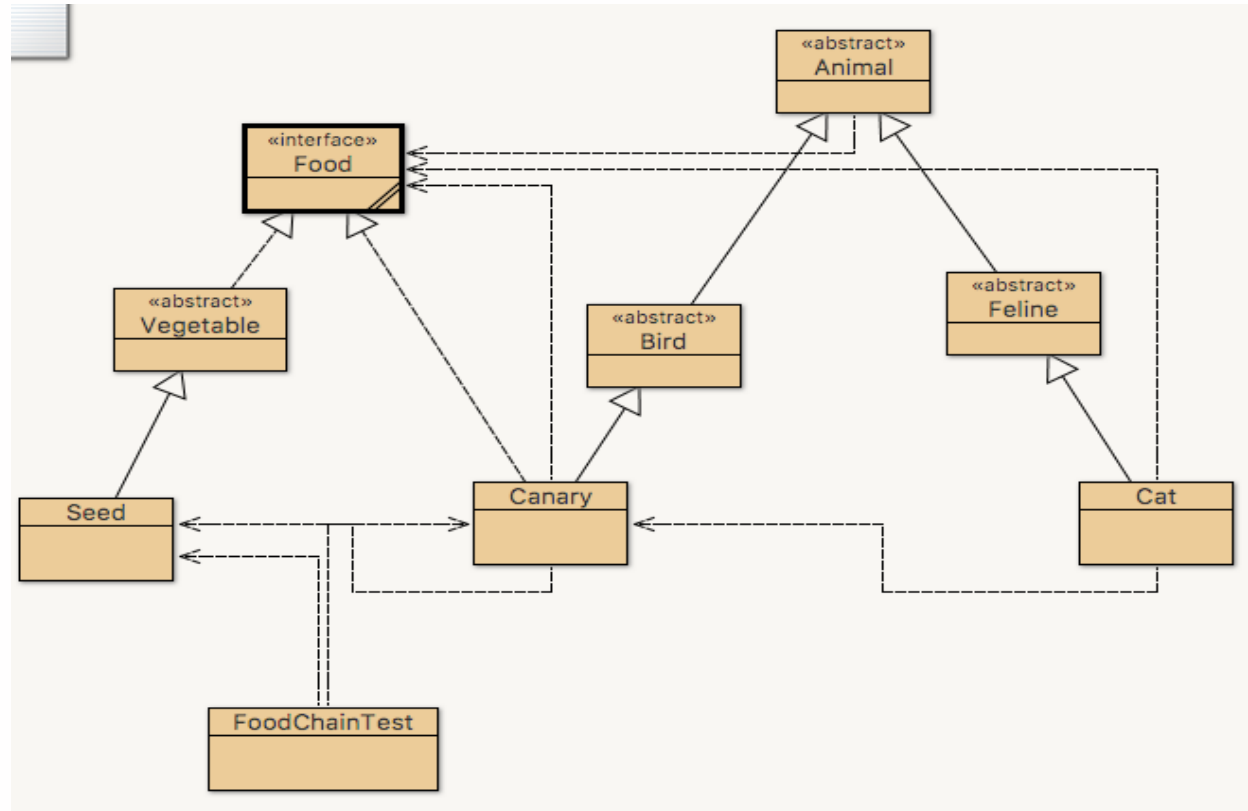
Canary will also be required to implement Foods two methods

```
public int getCalories(){  
    return getEnergy();  
}
```

```
public int extractEnergy(){  
    int cal = energy;  
    energy = 0; //should this Canary have status 'deceased'?  
    return cal;  
}
```



If you've followed these instructions, you should find that the *eat* method of Cat now compiles
A Canary is now a Food type as it implements the Food interface



Cat's eating problem solved

```
@Override
public boolean eat(Food food)
{
    if(food ==null){ // if the reference points to null
        return false; // immediately return. Method execution goes no further
    }

    if(food instanceof Canary){ // is food pointing to a Canary object?
        Canary canary = (Canary) food; // cast reference to a Canary type
        energy+=canary.extractEnergy(); // extract the Canary's energy
        //sing(); // cats don't sing
        return true; // return. Method execution goes no further
    }else{
        System.out.println("I cannot eat this type of food");
    }
    return false;
}
```



Test your code

- Write a new test method in the FoodChainTest class
- Call it testv2
- Write Code to execute the code instructions in the comments below (Reuse some of the code in the testv1 method)
- Execute the method in the main method
- Check that the output is as expected

```
public void testv2(){  
  
    //Create 3 seed objects  
    //Create a Canary object  
    //Have the Canary object eat first 2 seeds // should sing twice  
    //Create a Cat object  
    //Print out the Cat's energy // should be 0  
    //Have the Cat eat the 3rd seed  
    //Have the Cat eat the Canary  
    //Output the energy of the Cat //should be 20  
    //Output the energy of the Canary //should be 0  
    //Output the energy of the 3rd seed //should still be 10  
  
}
```



Interface vs Abstract class: **Similarities**

Similarities:

- Both can be used to provide '*templates*' for what subclasses can implement
- An abstract method plays the same role as an interface method – Both must be implemented in concrete form by a subclass
- An abstract class and an Interface can be used as the **type** for a reference variable.
E.g. `Food tasty = new Canary("tasty");`
- This code works **if** Food is an abstract class or Interface



Interface vs Abstract class: Differences

Differences:

- An abstract class is used for classic inheritance purposes – providing an abstract structure that subclasses inherit. The subclasses have a lot *in common*.
- E.g. the abstract class Bird provides common functionality for all feathered, winged animals

```
Bird canary = new Canary("mary");  
Bird ossie = new Ostrich("ossie");
```
- However, an interface is often used to impose common functionality on classes that have nothing in common.
- E.g. The interface Food imposes common (Food) functionality on two quite different classes : Seed and Canary

```
Food tasty = new Canary("tasty");  
Food sunflower = new Seed();
```



On the next slide, we compare the similarities and differences between the abstract class and interface versions of Food



```
public abstract class Food
{
    int calories; // abstract classes have fields

    /**
     * Abstract classes can have constructor
     */
    public Food()
    {
        calories = 0;
    }

    public abstract int getCalories();

    public abstract int extractEnergy();
}
```

VS

```
public interface Food
{
    // interfaces don't have fields
    //interfaces don't have constructors

    public int getCalories();//like an abstract method - but no abstract keyword

    public int extractEnergy();//like an abstract method - but no abstract keyword
}
```

Differences/Similarities: Syntax

- An abstract class has the term **abstract class** in its class declaration
- An interface has the term **interface** in its declaration
- An abstract class may have fields; an interface usually will not*
- An abstract class may have a constructor; an interface will not
- A class will use the keyword **extends** in its class declaration when inheriting from an abstract class
- A class will use the keyword **implements** in its class declaration to indicate that it will implement an interface
- A class can only extend one superclass (abstract or concrete). However, it can implement **multiple** interfaces
- An abstract class may have a concrete method; an interface will not
- An abstract method has the **abstract** keyword in its method declaration; an interface method does not
- An interface method and an abstract method do not have a method body

***When fields are declared in an Interface, they are public, static and, final by default**

We will not be covering examples with fields declared in Interfaces





OLLSCOIL NA GAILLIMHE
UNIVERSITY OF GALWAY

CT2106

Object Oriented Programming



Dr. Frank Glavin
Room 404, IT Building
Frank.Glavin@UniversityofGalway.ie
School of Computer Science

University
ofGalway.ie

OOP topics covered to date

Class structure - fields, constructor

Encapsulation

Instance methods

Object communication

Composition

OO design

Collections/ArrayLists/Arrays

Inheritance

Overriding methods

Class hierarchies

Polymorphism

Dynamic Dispatch

Abstract classes and methods

Interfaces



Topics not yet covered

Static methods

Private methods

Exception handling



Remaining weeks

Over the next few weeks, I am going to focus on getting you to apply the techniques you've already learned to solve different programming problems

This week we are going to look at creating a hierarchical data structure
In semester II, you'll be looking at more of these types of structures



Assembly

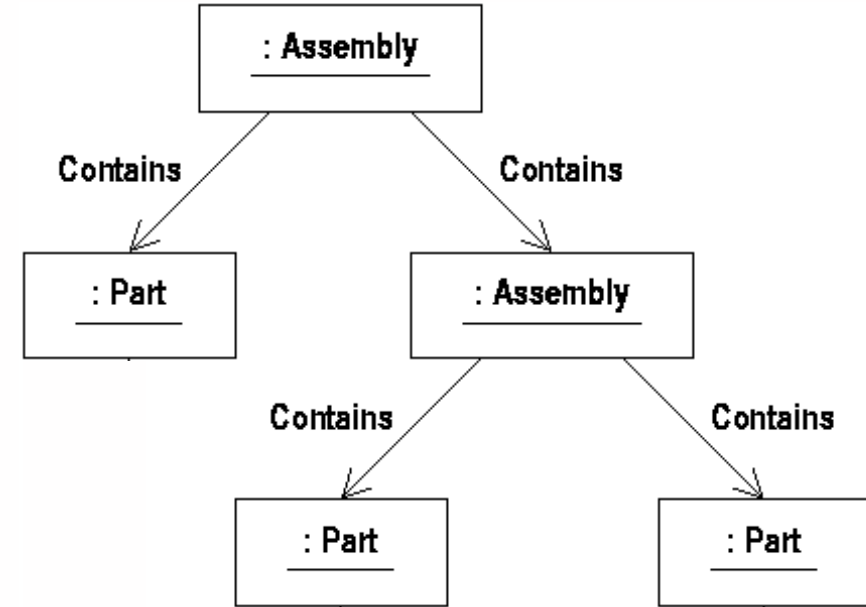
We want to design a data structure to keep track of the parts in a warehouse

Each part has a serial number, name and cost

Parts can be grouped together into an Assembly

An Assembly can hold other Assemblies as well as

Parts



Basic Classes

1. Part
2. Assembly

What is the relationship between a Assembly object and a Part object?

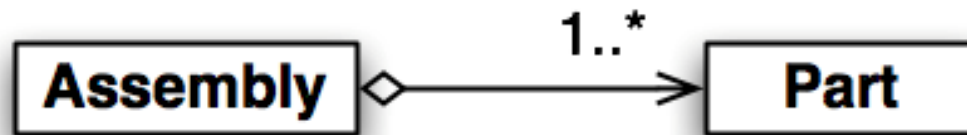


Basic Classes

1. Part
2. Assembly

What is the relationship between a Assembly object and a Part object?

Any Assembly object is **composed** of multiple Part Objects
In other words, Assembly object has a **has-a** relationship with Part



Is-a vs has-a relationships

Recall that there are two fundamental relationships between classes in OO

is-a (or inheritance)

has-a (or composition)

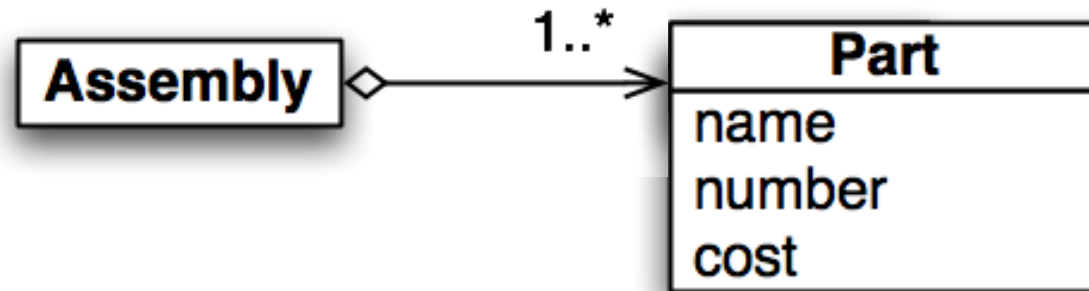
A RacingBike **is-a** type of Bicycle (inheritance)

A RacingBike **has-a** Wheel (Composition)



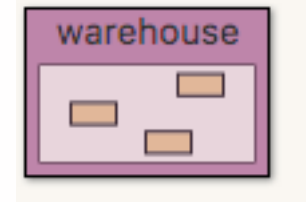
Part

- A Part object has the following properties
Name
ID number
Cost
- We can represent these as follows in a class diagram

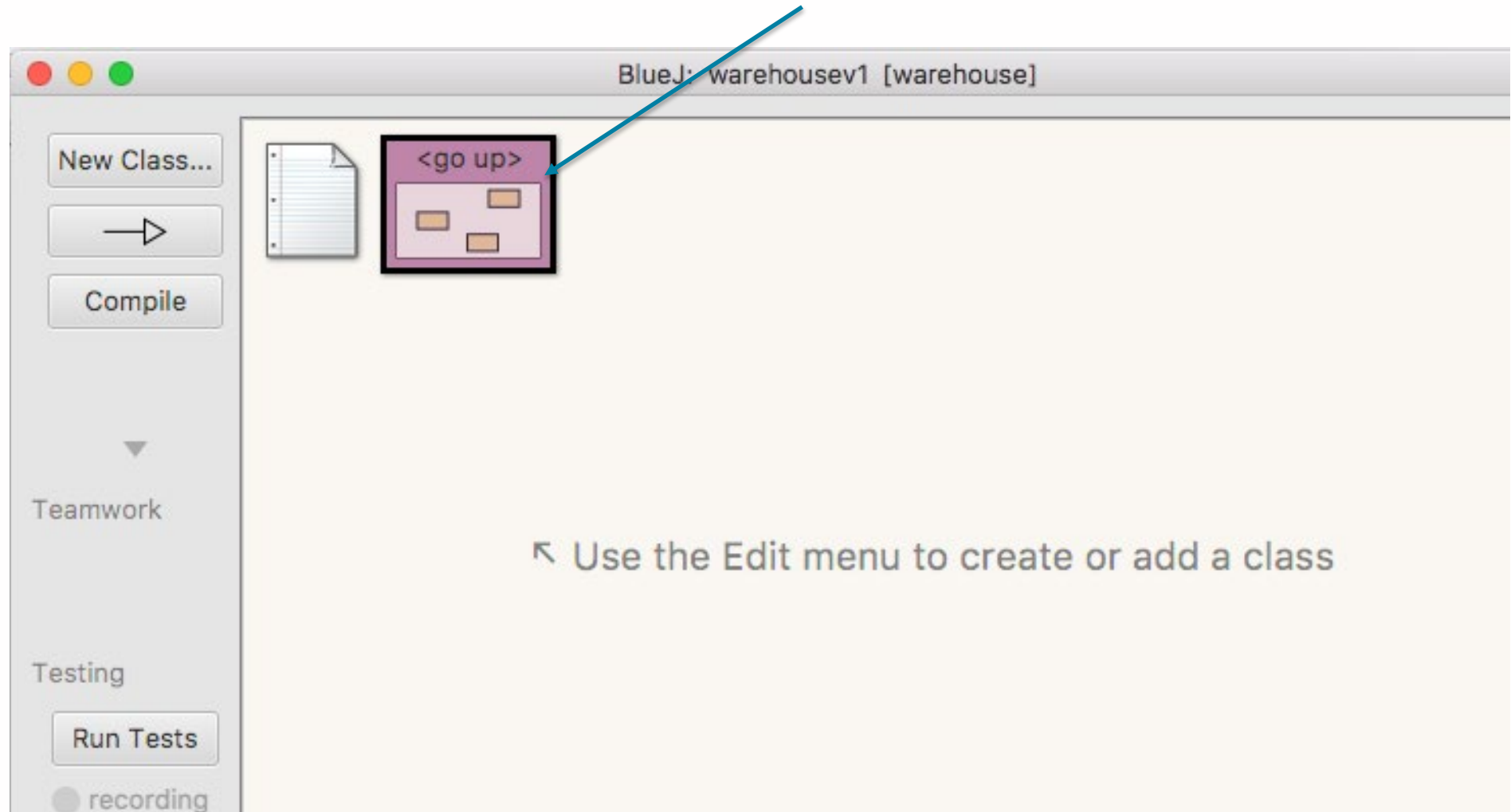


package

- Java organises groups of related classes into what is known as **packages**
- We are going to put all our Part-Assembly code into a package called **warehouse**
- **In BlueJ create a new Project**
- **In the BlueJ menu, Choose Edit->New Package**
- **Enter the name *warehouse***
- **Click on the package icon created**



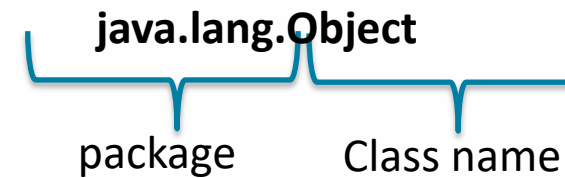
We will create our classes in the warehouse package



packages

- When a class is part of a package it has a **fully qualified name** : its name *and address*
- When you create a Part class in the package warehouse its fully qualified name will be **warehouse.Part**
- You've already encountered this:

`java.lang.Object`



A blue bracket underlines the text "java.lang.Object". A vertical line descends from the center of the bracket to the word "package". Another vertical line descends from the center of the bracket to the text "Class name".

package Class name

`java.util.ArrayList`



A blue bracket underlines the text "java.util.ArrayList". A vertical line descends from the center of the bracket to the word "package". Another vertical line descends from the center of the bracket to the text "Class name".

package Class name



Part

Part
name
number
cost

Now Write the Code for the **Part** class

Observe the guidelines on encapsulation

Decide what type your field variables should be

The constructor should initialise the fields with its input parameter values



Test Code

1. Now create a test class with a **main** method
2. In the main method, **create an array** of Part references, size 1000
3. **Create a loop** to place a reference to a new Part object in each location of the array.

E.g. each Part can have the following values:

name = "screw", number=28834, cost=0.02

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

```
//TODO
```

```
}
```



- Our program is required to hold multiple objects, say, of type Part
- Many Parts will have the same value
- Can you identify any problems with our implementation of Part?



Programing Principle: Avoid Data Replication

- All part objects of the same kind have the same attribute values (name, number, cost)
 - **Wasteful** of memory resources
 - **Hard to maintain** e.g. if the cost changes we have to change the cost in every object

myScrew : Part
name = "screw" number = 28834 cost = 0.02

myScrew : Part
name = "screw" number = 28834 cost = 0.02

myScrew : Part
name = "screw" number = 28834 cost = 0.02

myScrew : Part
name = "screw" number = 28834 cost = 0.02

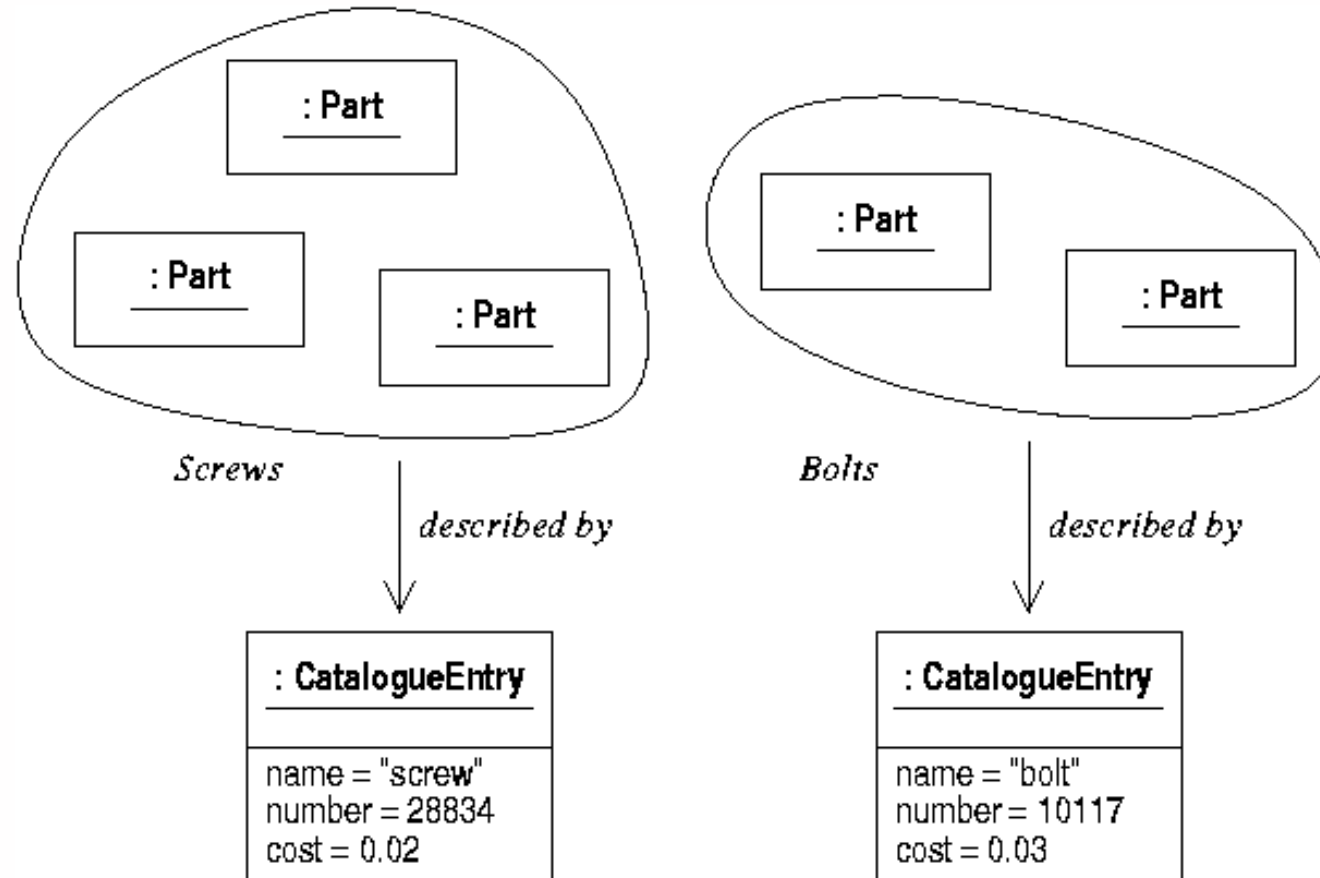


Avoiding Data Replication

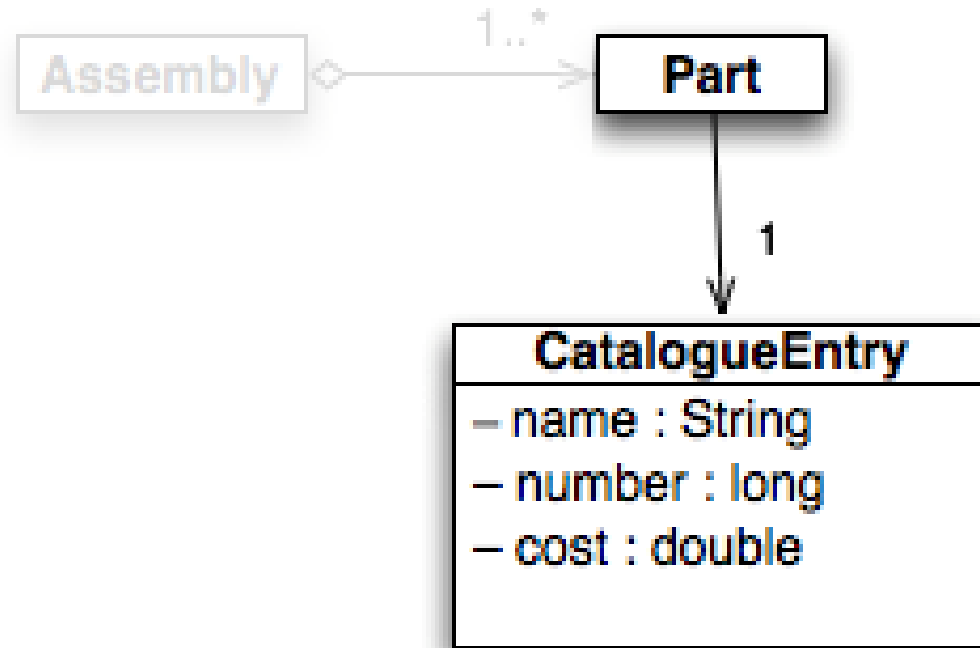
- Create a new class to store **shared information** about a particular Part
 - Call this a ‘catalogue entry’
 - Represents a catalogue entry that describes a type of part
 - Multiple parts of the same type are then described by one entry



All parts of the same type are linked to a single CatalogueEntry



Current model

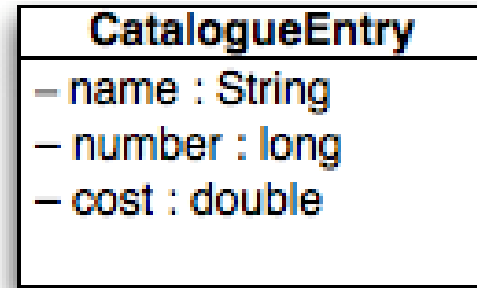


The class diagram tells us all we need to know to convert it into code



In BlueJ

Create a new Class called CatalogueEntry



It has the three fields as shown above

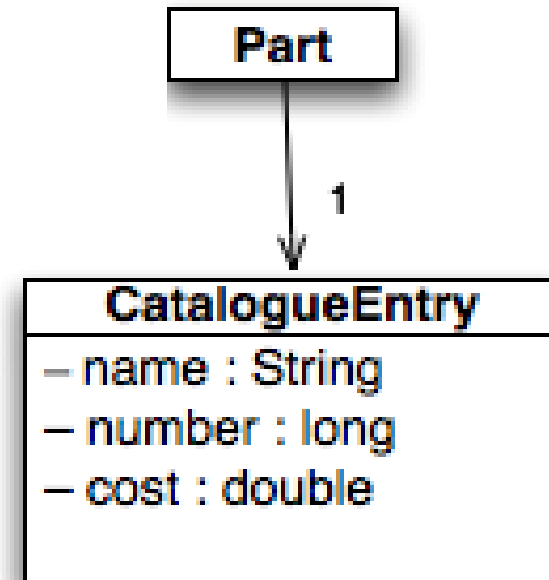
Observe the usual guidelines on encapsulation

The constructor should initialise these fields



Linking Part to CatalogueEntry

Each `Part` object should have a link to its corresponding `CatalogueEntry` object



Part class

Revise (refactor) your Part code

1. Remove the instance fields
2. Create a new field to hold a reference to a CatalogueEntry object
3. Refactor the Constructor so that it takes a CatalogueEntry object as a parameter
4. Revise your getter methods so that they call the relevant method from CatalogueEntry



Example Code

- Now revise your test code
- In the main method
 - Create a CatalogueEntry object of type “screw”, id number 28834, cost 0.02
 - Then use the CatalogueEntry object to create a 1000 Part objects



Revised Code?



OLLSCOIL NA GAILLIMHE
UNIVERSITY OF GALWAY

Review

- We've introduced a `CatalogueEntry` Class that holds the information about `Part` types
- When we create a `Part` of a certain type we use its corresponding `CatalogueEntry` object
- So multiple `Part` objects (of type 'nail'), all have links to a single `CatalogueEntry` object describing a nail
- The link between any nail `Part` and its `CatalogueEntry` object is implemented as an instance variable



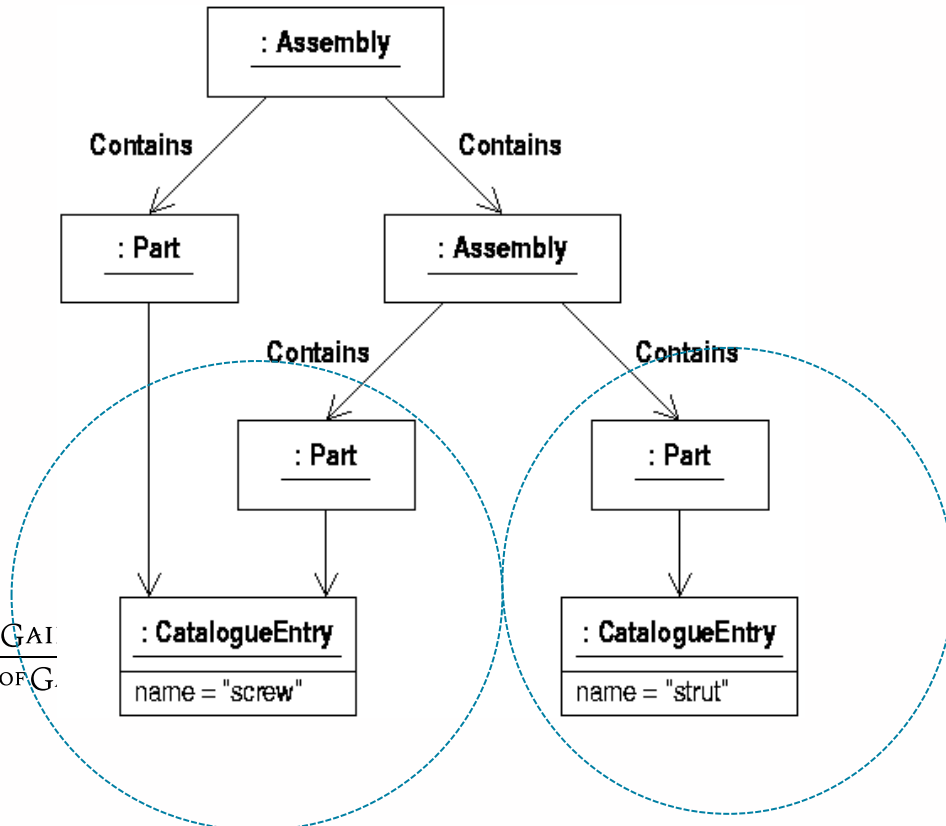
Composition

- Linking Part and CatalogueEntry is an example of **Object Composition**
- Object Composition refer to **constructing the functionality of an object by composing it from other objects.**



Stock Control Data Structure

- Assemblies should have a hierarchical structure
- i.e. An Assembly should hold other Assembly objects as well as Part objects



We've implemented these bits



OLLSCOIL NA GAI
UNIVERSITY OF G.

Implementing an Assembly

An **Assembly** needs to hold references to multiple Part objects
This is another example of **composition** – an object that is composed of other objects

We don't know in advance how many Part objects needed
How will we solve that?





OLLSCOIL NA GAILLIMHE
UNIVERSITY OF GALWAY

CT2106

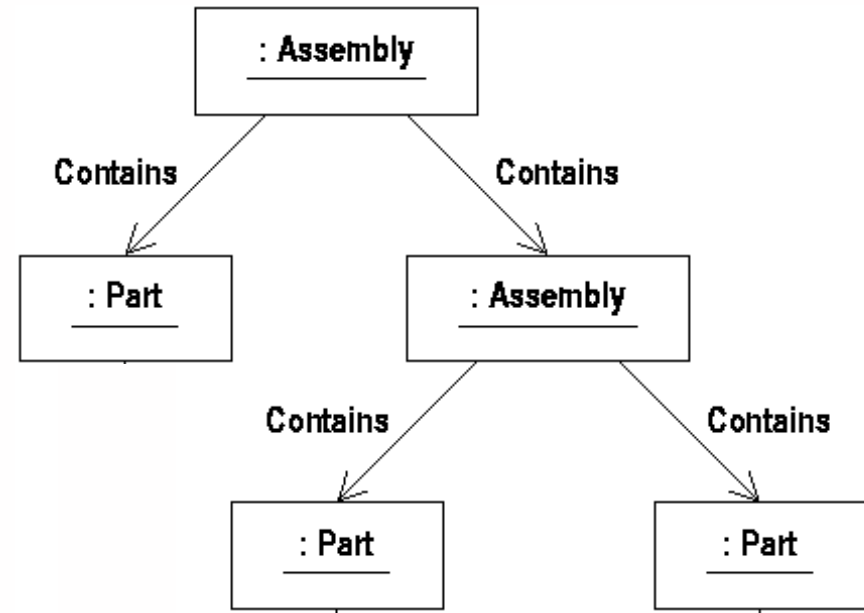
Object Oriented Programming



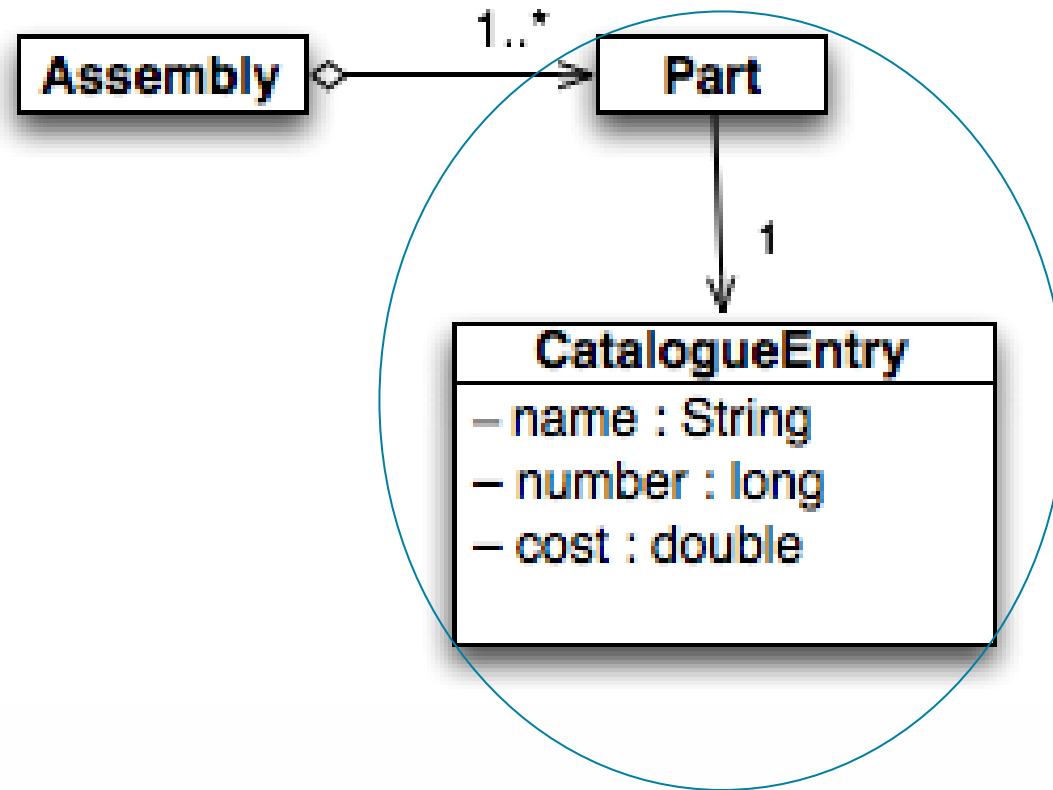
Dr. Frank Glavin
Room 404, IT Building
Frank.Glavin@UniversityofGalway.ie
School of Computer Science

University
ofGalway.ie

We want to design a data structure to keep track of the parts in a warehouse
Each part has a serial number, name and cost
Parts can be grouped together into an Assembly
An Assembly can hold other Assemblies as well as Parts



Yesterday, we left off here:



```
public void partTest(){  
    CatalogueEntry entry = new CatalogueEntry("nail", 2333445, 0.02);  
    Part[] parts = new Part[1000];  
    for(int i=0; i< parts.length; i++){  
        parts[i] = new Part(entry);  
    }  
}
```



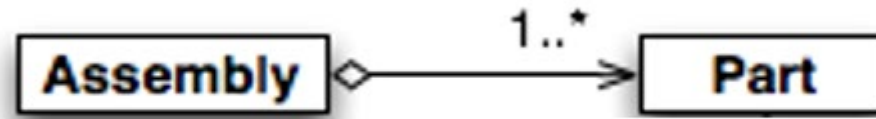
- You can continue using your own code from yesterday or you can download and add the zip file I posted in Week 9
- As usual, in BlueJ
Project -> Add Zip/Jar



Back to implementing an Assembly

An Assembly **is composed** of multiple Parts

We don't know in advance how many Parts it should hold



This suggests that we should use a dynamically resizable container like an ArrayList



Assembly

In the warehouse package

1. Create an Assembly class
2. It should have a private field *name* of type String
3. It should have a private field *parts* of type ArrayList. The ArrayList is meant to contain Part references. (Remember that you will need to use the import java.util.ArrayList statement)
4. Assembly should have an **add** method that allows a Part to be added to the Assembly
5. Assembly should have a **getCost** method that returns a double value – leave the implementation blank



Assembly class

The `Assembly` class has a private instance variable pointing to an `ArrayList` of `Part` references

The `add` method adds a `Part` object to the `ArrayList`

`getCost` returns the overall cost of the Parts in the Assembly

```
package warehouse;

import java.util.ArrayList;

public class Assembly
{
    // instance variables - replace the example below with your own
    private ArrayList<Part> parts = new ArrayList();
    private String name;

    /**
     * Constructor for objects of class Assembly
     */
    public Assembly(String name)
    {
        this.name = name;
    }

    /**
     * add method - replace this comment with your own
     * @param part : a reference to a Part to add
     * @return true if part was added successfully
     */
    public boolean add(Part part)
    {
        return parts.add(part);
    }

    public double getCost(){
        //TODO
        return 0;
    }
}
```



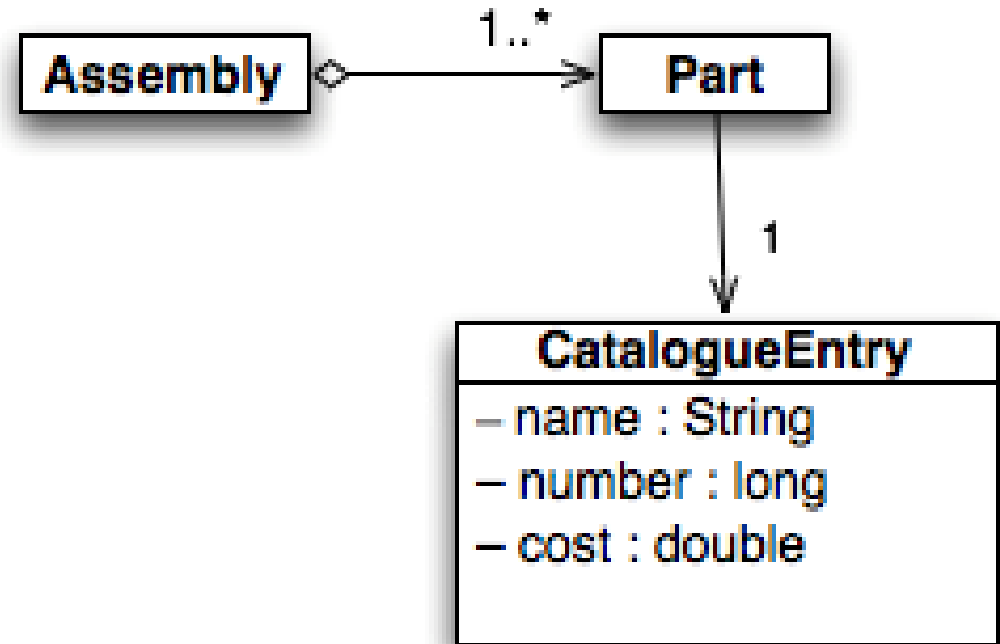
Status

We've created 3 classes:

Part

CatalogueEntry

Assembly



Reuse the Test Class from yesterday

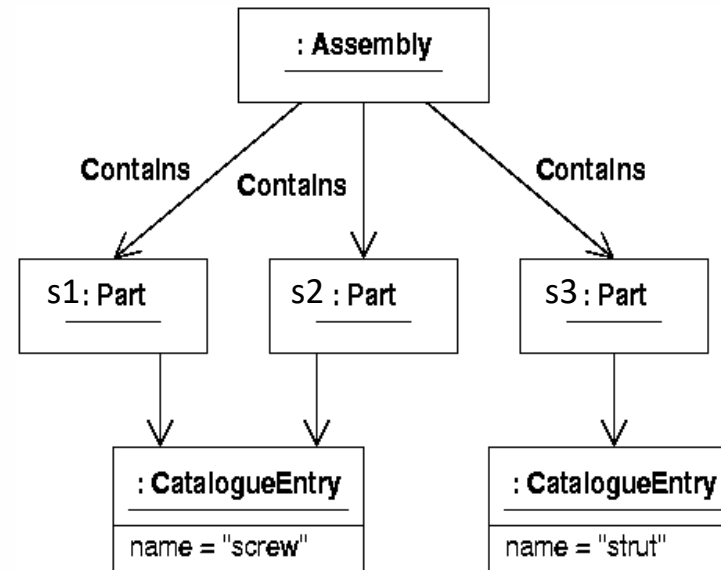
```
package warehouse;
/**
 * PartTest is used to write test code
 * for the Assembly Part classes.
 * @author (Ihsan Ullah)
 * @version (Nov 8th)
 */
public class AssemblyTest
{
    public void partTest(){
        CatalogueEntry entry = new CatalogueEntry("nail", 2333445, 0.02);
        Part[] parts = new Part[1000];
        for(int i=0; i< parts.length; i++){
            parts[i] = new Part(entry);
        }
    }
    public int costTest(){
        // TODO :Create the test code here
        return 0;
    }
    public static void main(String[] args)
    {
        // put your code here
        AssemblyTest assmblTest = new AssemblyTest();
        assmblTest.partTest();
        int value = assmblTest.costTest();
    }
}
```



Reuse the Test Class from yesterday

In the **costTest** method write code to implement the structure in the figure

1. Create an Assembly
2. Create two CatalogueEntry objects
3. Create 3 Parts of known cost
4. Add them to Assembly
5. Call the cost method of the Assembly to return the overall cost
6. If the Assembly returns the right answer, then our classes are working



Your code should look like this

```
public void costTest(){
    Assembly assembly = new Assembly("My First Assembly");
    CatalogueEntry catEntryScrew = new CatalogueEntry("screw", 12344455, 0.02);
    CatalogueEntry catEntryStrut = new CatalogueEntry("strut", 3455522, 0.05);
    Part s1 = new Part(catEntryScrew); // cost 0.02
    Part s2 = new Part(catEntryScrew); // cost 0.02
    Part s3 = new Part(catEntryStrut); // 0.05
    assembly.add(s1);
    assembly.add(s2);
    assembly.add(s3);
    double total= assembly.getCost(); // should return 0.09
    System.out.printf("total cost: %f", total);
}
```



Overall cost of an Assembly ?

The overall cost of an assembly is a sum of the cost of its Part objects. Thus the `getCost()` method for assembly needs a way of iterating over the `ArrayList` and calling the `getCost()` method of each Part

```
public double getCost(){  
    //TODO  
    return 0;  
}
```



Implement the `getCost()` method

You may use the comments below to guide you

```
public double getCost(){  
  
    // define a variable of type double called totalCost  
    // the initial value of totalCost = 0  
    // for each Part in the parts ArrayList  
    //     call its getCost() method  
    //     add the value returned to totalCost  
    // return the value of totalCost  
    return 0;  
}
```



getCost () method

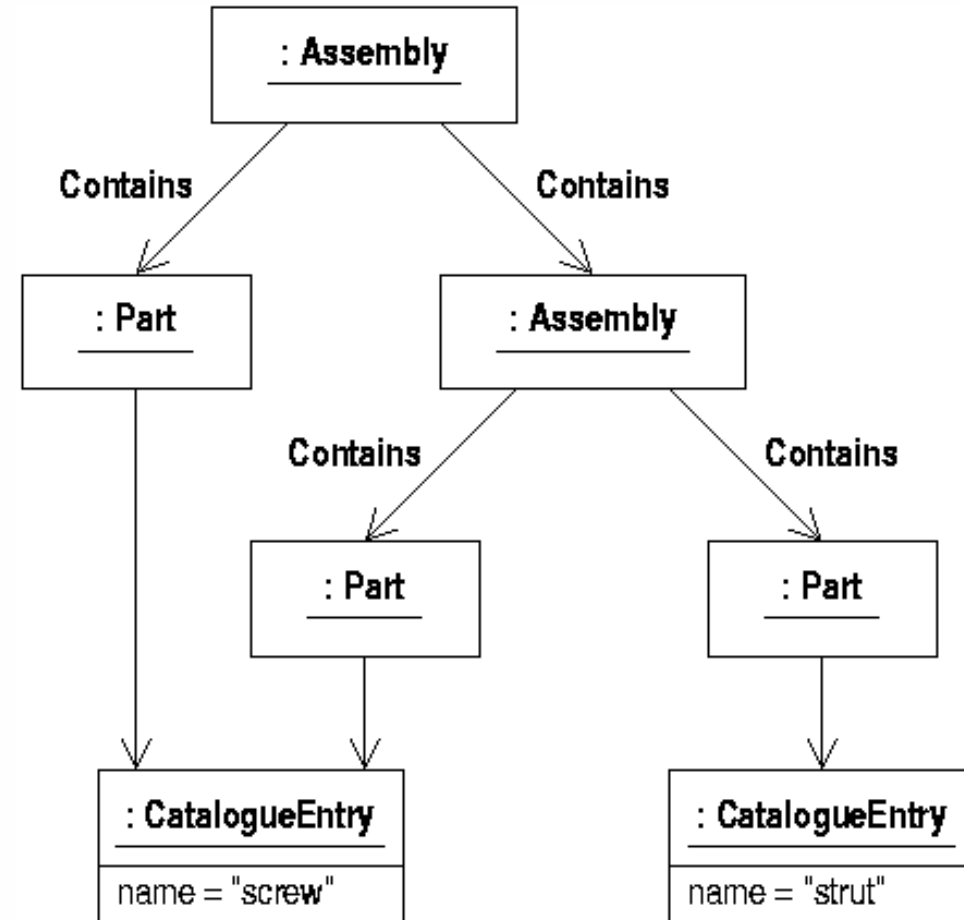
```
/**
 * getCost method
 * returns the total cost of all items in the Assembly
 */
public double getCost(){
    double totalCost = 0;
    for(Part part: parts){
        totalCost+=part.getCost();
    }
    return totalCost;
}
```

We used the reduced loop syntax to iterate over the ArrayList



Subassemblies

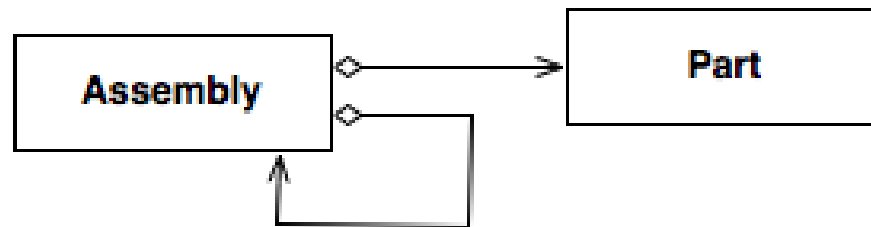
An `Assembly` object should be able to contain `Parts` **AND other `Assembly`** objects



An Initial Solution

Any Assembly would be composed of other Assemblies and Parts

Suggestion of an approach?



Initial Solution?

An arraylist that contains other Assembly objects

```
public class Assembly
{
    private ArrayList<Part> parts = new ArrayList();
    private ArrayList<Assembly> assemblies = new ArrayList();
    private String name;
}
/++
```



- We'd need to create a new add method for Assembly objects
- In other words, we create another version of add
- We **overload** the add method

```
public boolean add(Part part)
{
    return parts.add(part);
}
```

```
public boolean add(Assembly assembly)
{
    return assemblies.add(assembly);
}
```



We'll need also to create a new cost method for Assembly objects

```
public double getCost(){  
    double totalCost = 0;  
  
    for(Part part: parts){  
        totalCost+=part.getCost();  
    }  
  
    for(Assembly assembly: assemblies){  
        totalCost+=assembly.getCost(); // this is a recursive call  
    }  
  
    return totalCost;  
}
```

Note : this is a recursive call



Any objections to this approach?



- From an OOP perspective - this is an awful solution
- Large amount of code repetition and redundancy

```
private ArrayList<Part> parts = new ArrayList();  
private ArrayList<Assembly> assemblies = new ArrayList();
```

```
public boolean add(Part part)  
{  
    return parts.add(part);  
}  
  
public boolean add(Assembly assembly)  
{  
    return assemblies.add(assembly);  
}
```

```
public double getCost(){  
    double totalCost = 0;  
  
    for(Part part: parts){  
        totalCost+=part.getCost();  
    }  
  
    for(Assembly assembly: assemblies){  
        totalCost+=assembly.getCost(); // this is a recursive call  
    }  
  
    return totalCost;  
}
```



Problems?

- **It is not extensible** – for example, let's say I wanted to add a new type of object to an Assembly
- Let's call it **Service** – representing 'After Sales Service'
- I would have **to completely rewrite and recompile the Assembly class**

New ArrayList to hold Service objects

New add method for Service objects

Another loop required in the getCost() method



Implications of Bad Design

Too many ArrayLists – one for each type

```
public class Assembly {  
  
    private ArrayList<Part> parts;  
    private ArrayList<Assembly> assemblies;  
    private ArrayList<Service> services;  
  
    public Assembly(){  
        parts = new ArrayList<Part>();  
        assemblies = new ArrayList<Assembly>();  
        services = new ArrayList<Service>();  
    }  
}
```



Implications of Bad Design

Code bloat : 3 overloaded add methods

```
public void add(Service service){
    services.add(service);
}

public void add(Part part){
    parts.add(part);
}

public void add(Assembly assembly){
    assemblies.add(assembly);
}
```



Implications of Bad Design

Unnecessary complexity

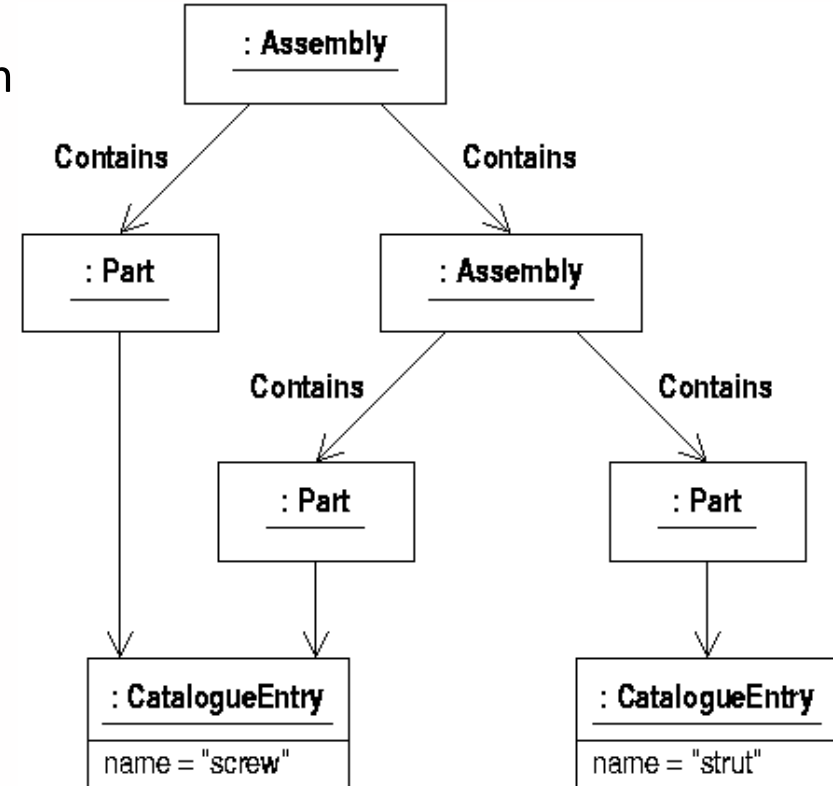
```
// would this work?  
public double cost(){  
    double cost = 0.0;  
    for(Part part : parts){  
        cost+=part.cost();  
    }  
    for (Service srv: services){  
        cost+=srv.cost();  
    }  
    for(Assembly assembl : assemblies){  
        cost+=assembl.cost();  
    }  
    return cost;  
}
```



Solution v2

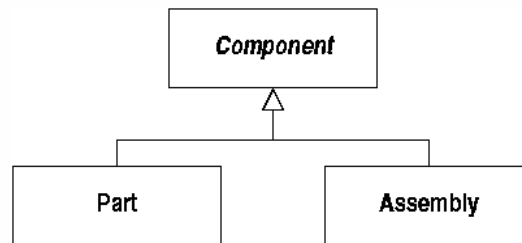
We can achieve an **elegant, extensible and concise** solution using two features of OO programming

1. Abstract classes/ Interfaces
2. Polymorphism



Solution

- The key is to make an **abstract class** or **interface** called `Component` (the name is not important)
- `Part` and `Assembly` should extend/implement `Component`



- Create an interface called `Component` with a single method `getCost ()`
- `Part` and `Assembly` should implement `Component`



Component

- An abstract class or interface with a single method `getCost ()`
- It can never be instantiated as an object
- But it can be used to make (polymorphic) references to its subclasses

```
public interface Component
{
    public double getCost();
}
```

or

```
public abstract class Component
{
    public abstract double getCost();
}
```



```
public class Part implements Component
{
    //every Part has a reference to a CatalogueEntry object
    private CatalogueEntry entry;

    public Part(CatalogueEntry e)
    {
        entry = e;
    }

    public String getName(){
        return entry.getName();
    }

    public long getNumber(){
        return entry.getNumber();
    }

    @Override
    public double getCost(){
        return entry.getCost();
    }
}
```



Assembly

- Each `Assembly` object should be able to hold multiple `Component` objects
 - Some of these will be `Part` objects
 - Some will be `Assembly` objects
- But as far as each `Assembly` object is concerned, it is holding a collection of `Component` objects



Refactoring Assembly

Four minor changes required

1. Add `implements Component` to class definition
2. Change `ArrayList` declaration so that it holds `<Component>` types
3. Remove the `add(Part)` and `add(Assembly)` methods and replace with a single `add(Component)` method
4. Modify the `getCost()` method so that it calls the `getCost()` method of the `Component` type



```
package warehouse;
import java.util.ArrayList;
```

1. Implements Component

```
public class Assembly implements Component
```

2. Type declaration changed to
<Component>

```
{
    private ArrayList<Component> components = new ArrayList();
    private String name;
```

```
    public Assembly(String name)
    {
        this.name = name;
    }
```

```
    public boolean add(Component component)
    {
        return components.add(component);
    }
```

3. existing add() methods
replaced with a single
add(Component) method

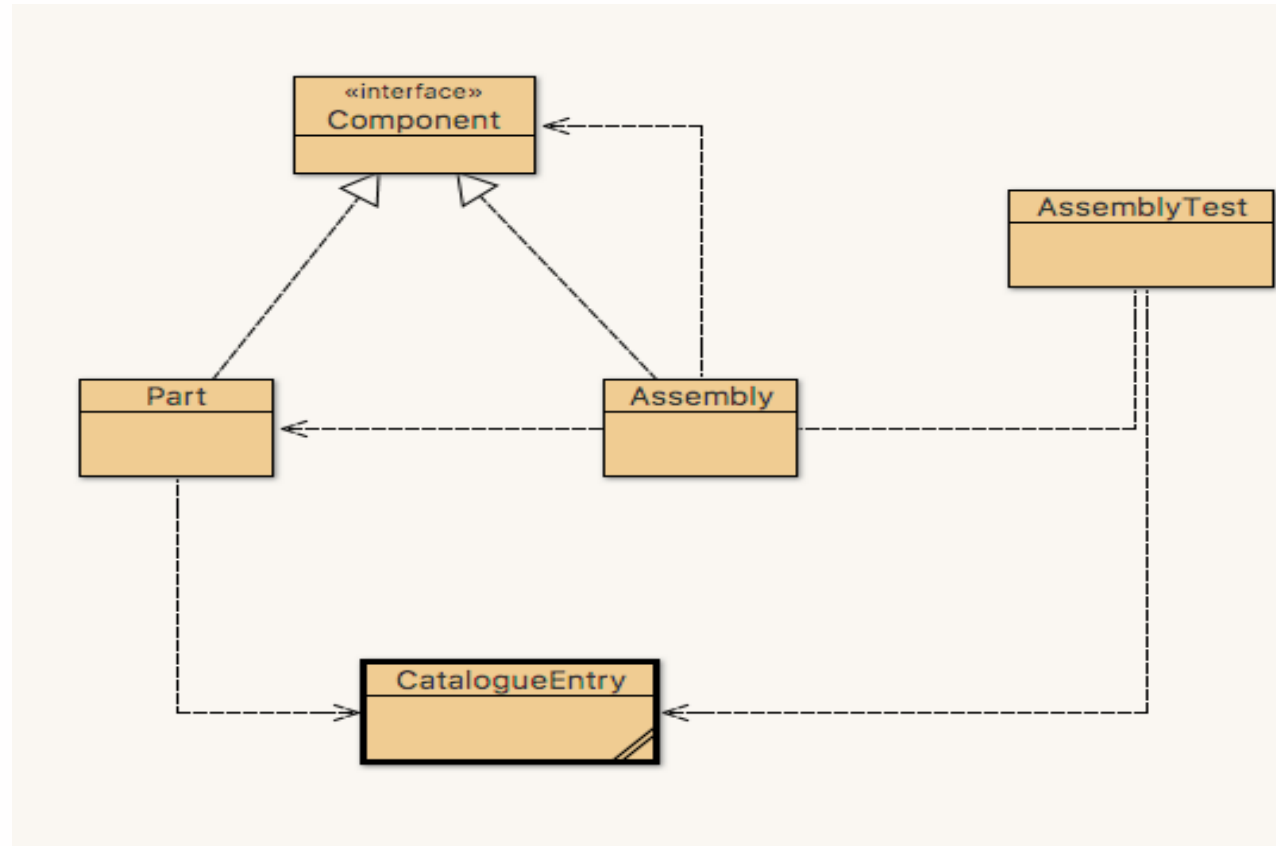
```
    @Override
    public double getCost(){
        double totalCost = 0;
        for(Component comp: components){
            totalCost+=comp.getCost();
        }
        return totalCost;
    }
```

4. getCost() method of
Component invoked

```
    public String getName(){
        return name;
    }
}
```



Rearrange your class diagram



Assembly has an 'is-a' relationship to Component
Component has 'has-a' relationship to Assembly



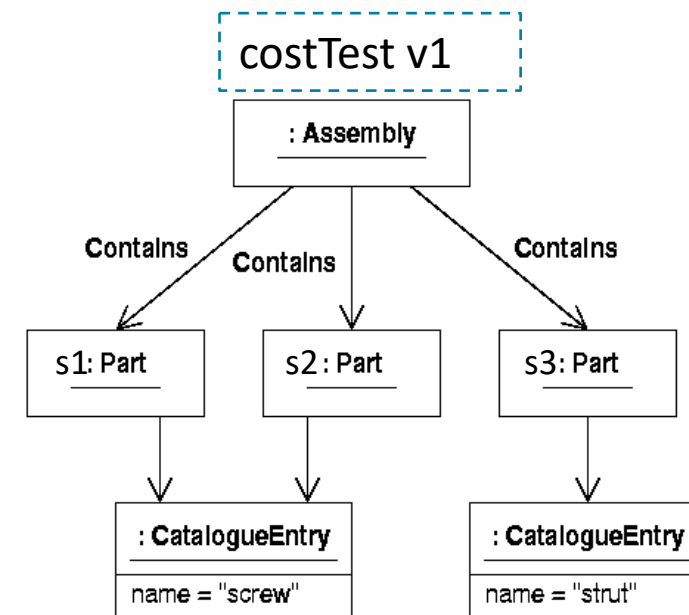
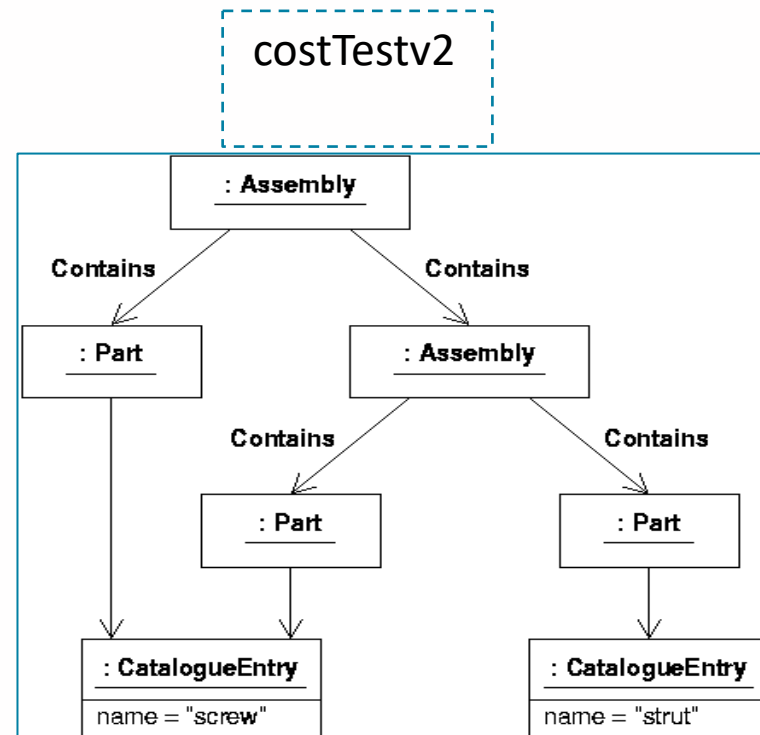
Compare the solutions

- Compare this version of Assembly to the bloated version we created earlier
 - 50% Less Code
 - Easier to understand
 - Extensible
- If I want to create a new Service class, I can create it simply by **implementing Component**
- Assembly will accept any object that is of type Component
- Thus, I can extend the range of data types that Assembly can handle without touching its code
- Just as long as each class implements Component

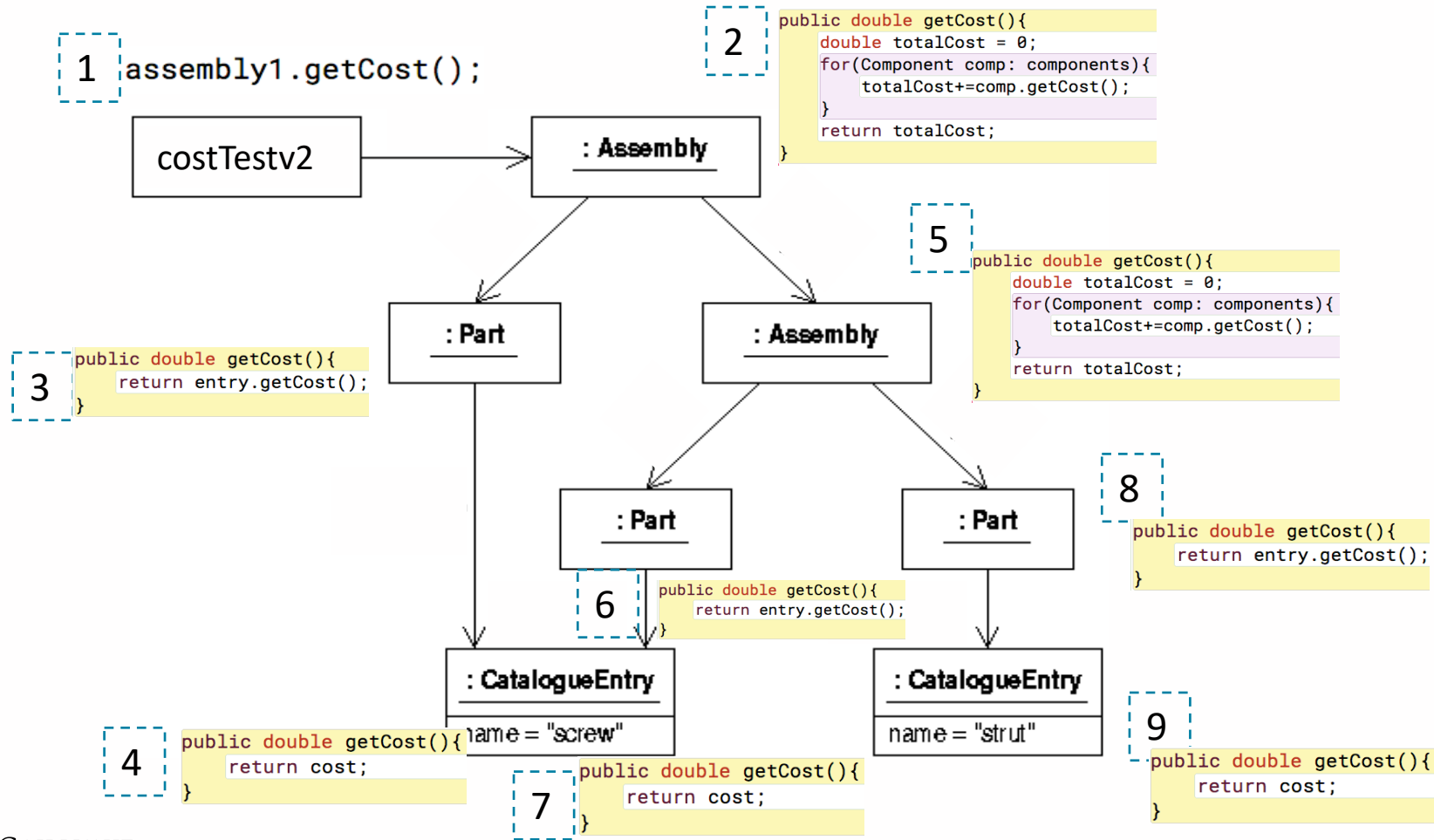


Creating a test method

- Create a new costTest Method – call it costTestv2
- Now, reuse the code you wrote for costTest v1 to represent the data structure on the left



What happens when we call the top Assembly object's `getCost()` method



Recursion

- Every reference to a `Component` object may be a reference to a `Part` or another `Assembly` object, whereby `getCost()` will be called again
- For each `Component` that is an `Assembly` object, its own `getCost()` is called
- This means that the `getCost()` method in the `Assembly` class is **recursive**
- The termination point is when all the `Part` objects within a particular `Assembly` have been encountered and the costs returned.
- The recursive nature of `getCost()` is *enabled by polymorphism*



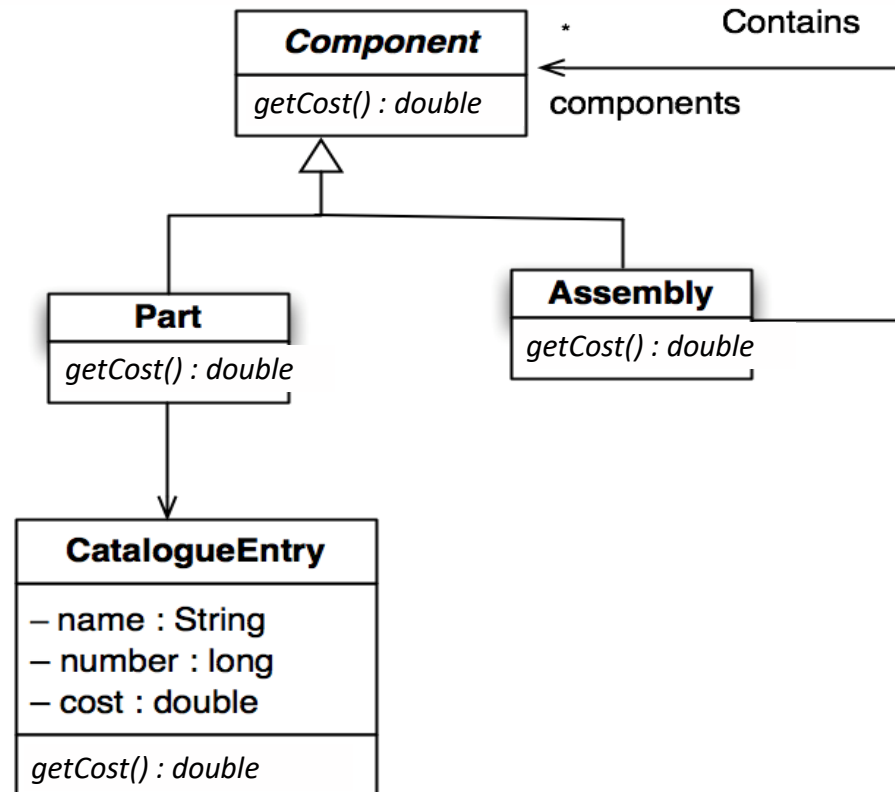
Recursion

- I will attach a few extra slides on recursion for you to look at.
- While the idea is easy, it is sometimes hard to grasp how a method executes a recursive call.
- While not an OOP concept per se, recursion is commonly used in algorithm and data structure design, so it is worth acquainting yourself with the idea



Composite

This data structure is based on a design pattern called **composite**



Lecture Wrap Up

- We've looked at creating a solution to the Assembly-Part problem
- Version 1 used the most obvious solution – storing Assembly references in another ArrayList
- This solution was inelegant, used more code than necessary – but more importantly, it could not be extended.
- Using an interface to link Part and Assembly into one type, Component, we were able to create a much simpler and extensible approach.
- The solution is an implementation of common OOP design pattern called “*Composite*”





OLLSCOIL NA GAILLIMHE
UNIVERSITY OF GALWAY

CT2106

Object Oriented Programming



Dr. Frank Glavin
Room 404, IT Building
Frank.Glavin@UniversityofGalway.ie
School of Computer Science

University
ofGalway.ie

Lecture Topics

Object-Oriented Design Patterns

Composite Design Pattern

Solving a problem with the *Composite* design pattern



Using BlueJ to quickly test code

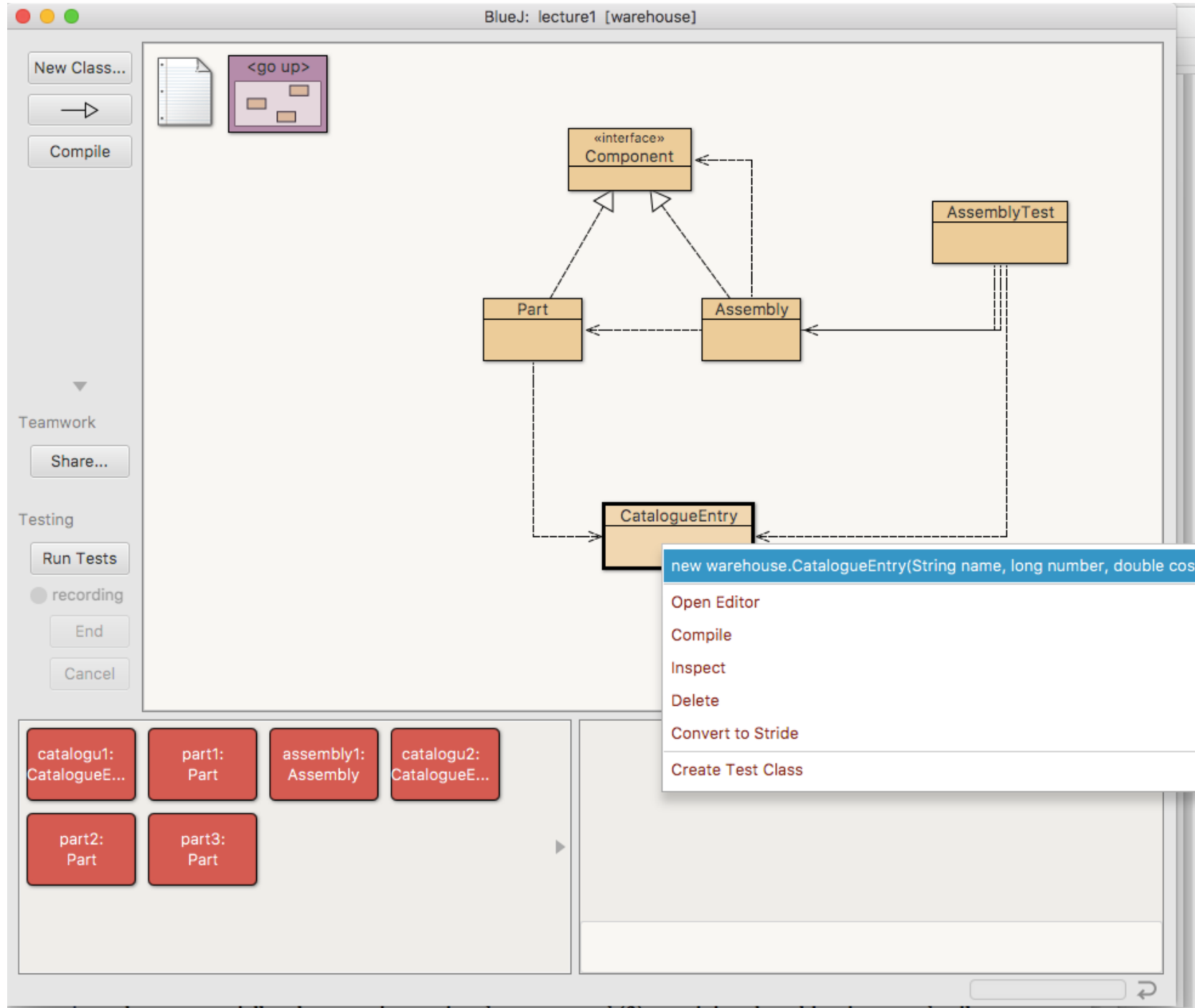
Download and add the jar file provided in the **Week 10 folder**



Reminder you can use BlueJ to quickly check your code.

Not a formal test by any means – just a sanity check





BlueJ: lecture1 [warehouse]

New Class...
 →
 Compile

Teamwork
 Share...

Testing
 Run Tests
 ● recording
 End
 Cancel

```

classDiagram
    class Component {
        <<interface>>
    }
    class Part
    class Assembly
    class CatalogueEntry
    class AssemblyTest

    Component <|-- Part
    Component <|-- Assembly
    Assembly ..> Part
    CatalogueEntry ..> Part
    CatalogueEntry ..> Assembly
    AssemblyTest ..> Assembly
    AssemblyTest ..> CatalogueEntry
  
```

catalogu1: CatalogueE...
 part1: Part
 assembly1: Assen...
 catalogu2:
 part2: Part
 part3: Part

assembly1 : Assembly

- inherited from Object
- boolean add(Component)
- double getCost()
- String getName()
- Inspect
- Remove



BlueJ: lecture1 [warehouse]

New Class...
→
Compile

Teamwork
Share...

Testing
Run Tests
recording
End
Cancel

«interface»
Component

Part

Assembly

AssemblyTest

BlueJ: Method Result

double getCost()

assembly1.getCost() returned:

double 0.08

Inspect
Get
Close

catalogu1: CatalogueE...
part1: Part
assembly1: Assembly
catalogu2: CatalogueE...

part2: Part
part3: Part

assembly1: Assembly



This is the `getCost` method belonging to `Assembly`. Explain why this method does not have to distinguish between `Part` and `Assembly` objects to return its overall cost.

1. The method contains `Component` objects, not `Part` and `Assembly` objects. Therefore, it doesn't call their methods.
2. Each `Component` reference is actually a polymorphic reference to a `Part` or `Assembly` object. Polymorphism ensures that when `getCost` is called on a component reference, the relevant `getCost()` method will be called on the referenced `Part` or `Assembly` object.
3. Each `Component` object is composed of a `Part` or `Assembly` object. When `getCost()` is called on a `Component` object it then calls the `getCost()` method of the `Part` or `Assembly` object it contains.
4. Each `Component` is encapsulated by a `Part` or `Assembly` Interface. This means that `Component` will implement the correct `getCost` method of `Part` or `Assembly`.

```
@Override
public double getCost(){

    double totalCost = 0;

    for(Component comp: components){
        totalCost+=comp.getCost();
    }

    return totalCost;
}
```



Identify where Recursion occurs in this method?

1. When the method loops through each component object adding up the total cost.
2. When the Part method getCost method calls the CatalogueEntry object.
3. Each time a Component reference is used to call the getCost() method on an Assembly Object.
4. When the method returns 0.



For some reason, you are asked to write code to allow a Canary object to be added an Assembly object. The Canary code is as follows. What changes do you make to allow a Canary object to be added to an Assembly.

1. public class Canary<Component> extends Bird
2. public class Canary extends Component
3. public class Canary extends Bird implements Component
4. Add a concrete implementation of the getCost() method as defined by the component interface
5. Option number 3 AND option number 4

```
public class Canary extends Bird
{
    private String name;
    private double cost;

    /**
     * Constructor for objects of class Canary
     */
    public Canary(String name)
    {
        this.name = name;
        cost = 5;
    }
}
```



OOP Design Principle: Open-Close Design Principle

*“Software entities like classes, modules and functions should be **open for extension but closed for modifications.**”*

This may seem counter-intuitive at first reading

Design your code so that it can be extended, and any extensions require the minimum of modification to your existing code



Adding a new Component class

Even though the `Assembly` class is closed for modification, I can still extend its functionality

```
public class Canary extends Bird implements Component
{
    private String name;
    private double cost;
    /**
     * Constructor for objects of class Canary
     */
    public Canary(String name)
    {
        this.name = name;
        cost = 5;
    }

    @Override
    public double getCost(){
        return cost;
    }
}
```



Canary as a Component

The key idea is that the Assembly object doesn't view the Canary object as a Type of Canary

It is just another Component with its own getCost method

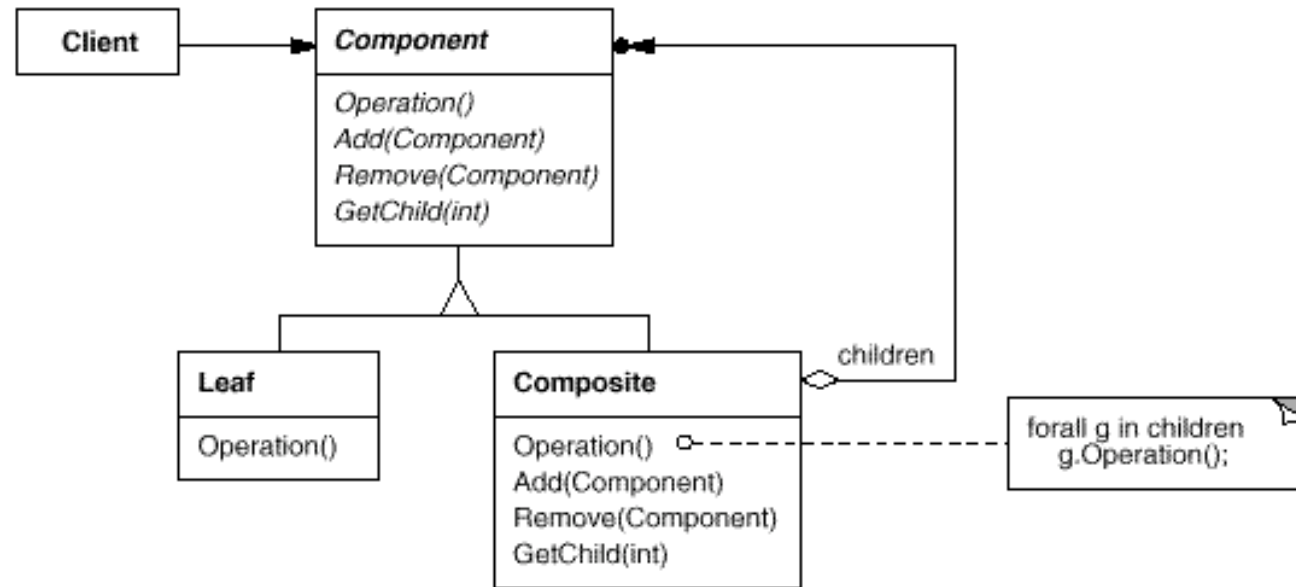
```
Canary someCanary = new Canary("Trumper");  
assembly.add(someCanary);
```



Composite Design Pattern

- This data structure is in fact a well known object oriented **design pattern**
 - the **Composite design pattern**
- Used to implement hierarchical data structures
- For example, directory/file structures





Purpose

Facilitates the creation of object hierarchies where each object can be treated independently or as a set of nested objects through the same interface.

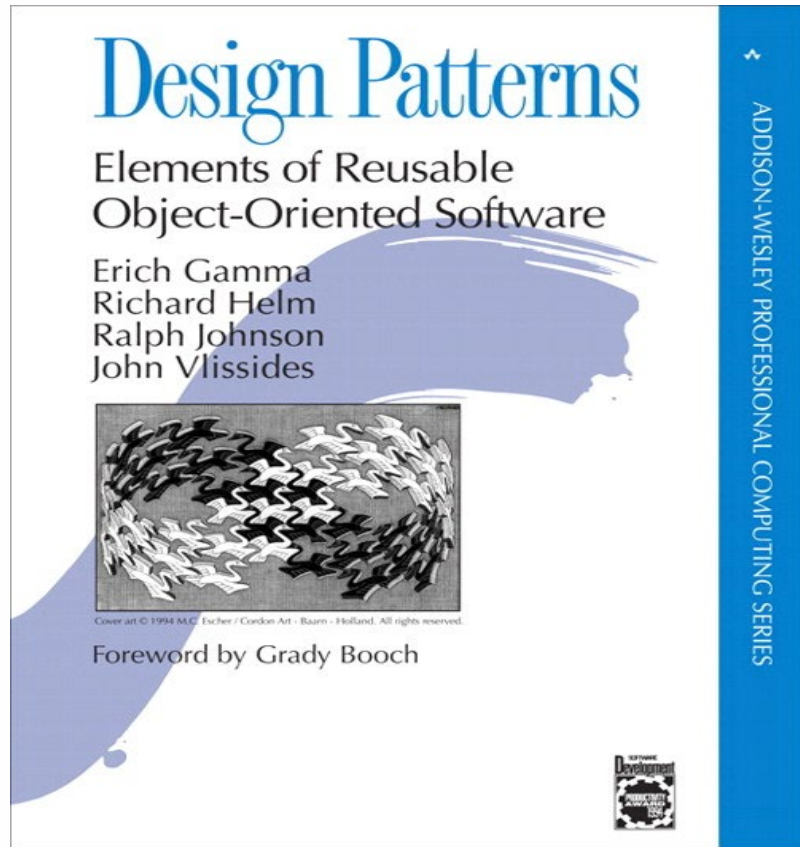
Use When

- Hierarchical representations of objects are needed..
- Objects and compositions of objects should be treated uniformly.



Design Pattern

A solution to a particular recurring design issue in a particular context:



“Each pattern describes a problem that occurs over and over again in our environment, and then describes the core of the solution to this problem in such a way that you can use this solution a million times over, without ever doing it the same way twice”

Erich Gamma et al., *Design Patterns*, 1995

Design Patterns

In general, a design pattern consists of:

- a *name*, for easy reference

- a motivation of the *problem* being solved

- a description of the *solution* proposed

- a discussion of the *consequences* of adopting the pattern



Rationale for Design Patterns

- Capture the knowledge of experienced developers
- Provide a publicly available “repository” of patterns
- Newcomers can learn these and apply them to their design
- Yields a better structure of the software (modularity, extensibility)
- Facilitates a common pattern language for discussions between programmers
- Facilitate discussions between programmers and managers



File System

- As mentioned before in lectures, the composite approach can be used to model the directory/file structure we have in our computers
- We will work through an exam question from a few years back



2. Read the scenario below and answer the questions that follow.

As a developer you are asked to write the Java code for a simple file system. The file system should be able to handle folders and files. Each folder can contain files of different types as well as other folders. Figure 2.1 illustrates an example of a folder structure your code has to be able to handle.

There are a number of operations that need to be implemented on the file system:

- **size:** returns the size in bytes of the files and folders contained in any one folder
- **numFolders:** returns the number of sub-folders in any folder
- **numFiles:** returns the number of files in any folder, including its subfolders
- **search:** searches the folder and its subfolders for a particular file using its name

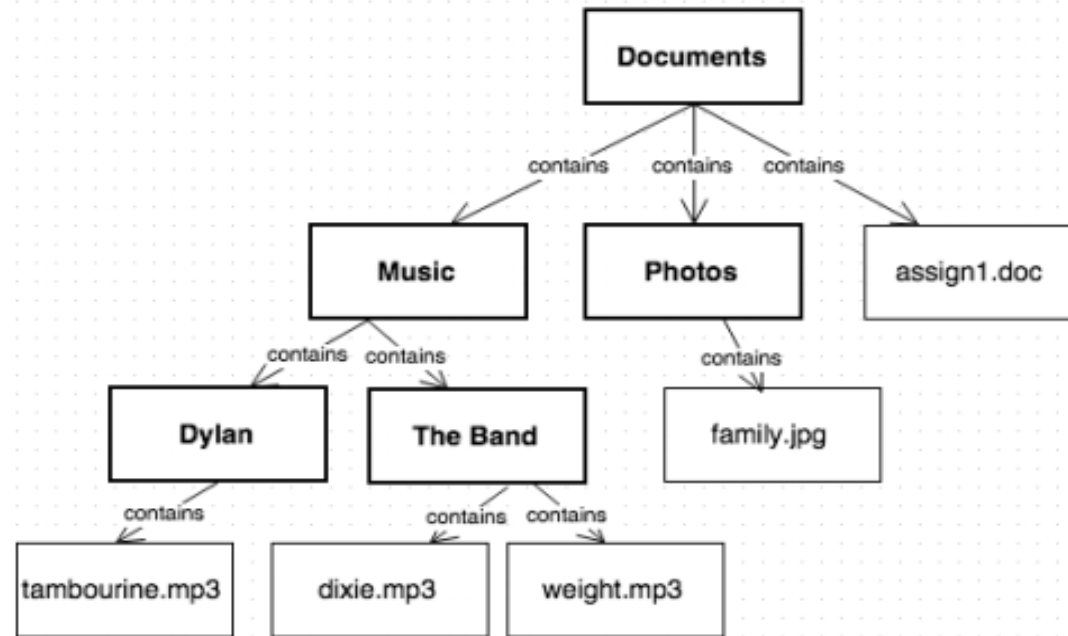


Figure 2.1: an example of a nested folder structure



Simple File System

- You are asked to write the Java code for a simple file system.
- The file system should be able to handle folders and files. Each folder can contain files of different types as well as other folders.

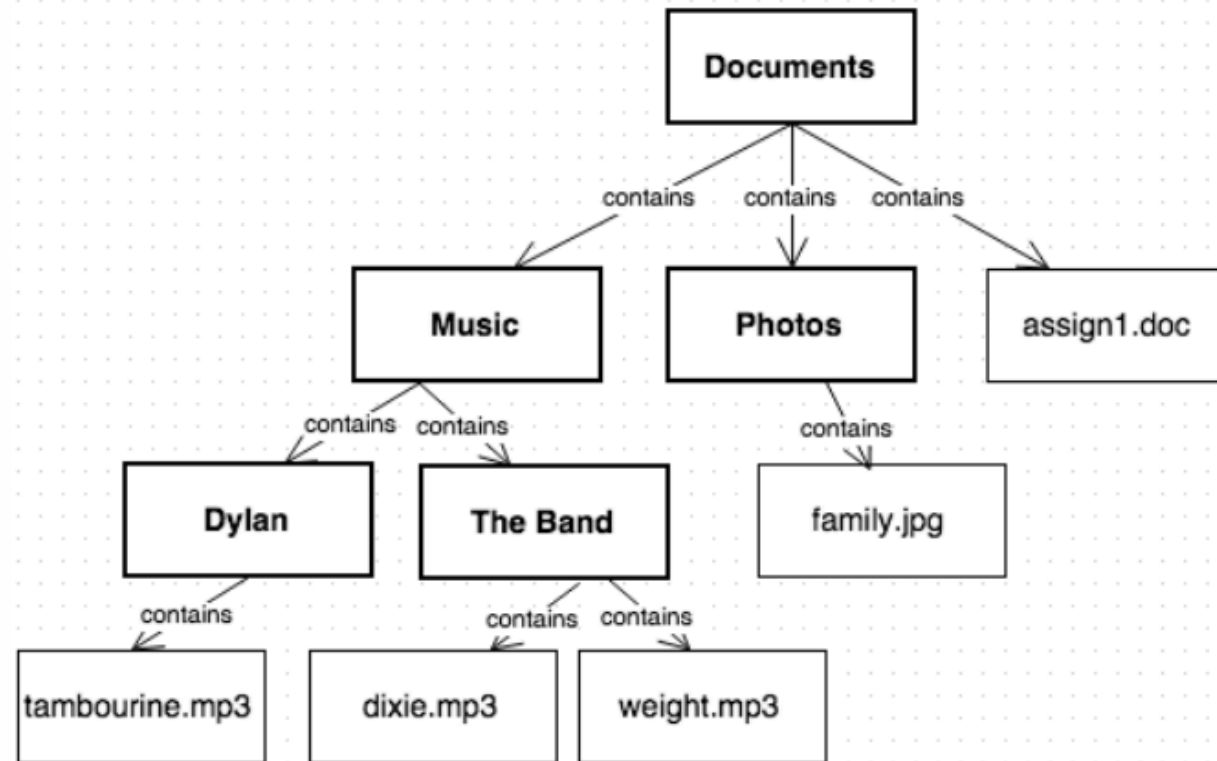


Figure 2.1: an example of a nested folder structure



Simple File System

You should be able to request the following from any folder

size(): returns the overall size (e.g. in bytes) of the files and sub-folders contained in any one folder

numFiles(): returns the number of files in any folder, including those in its sub-folders

numFolders(): returns the number of sub-folders in any folder

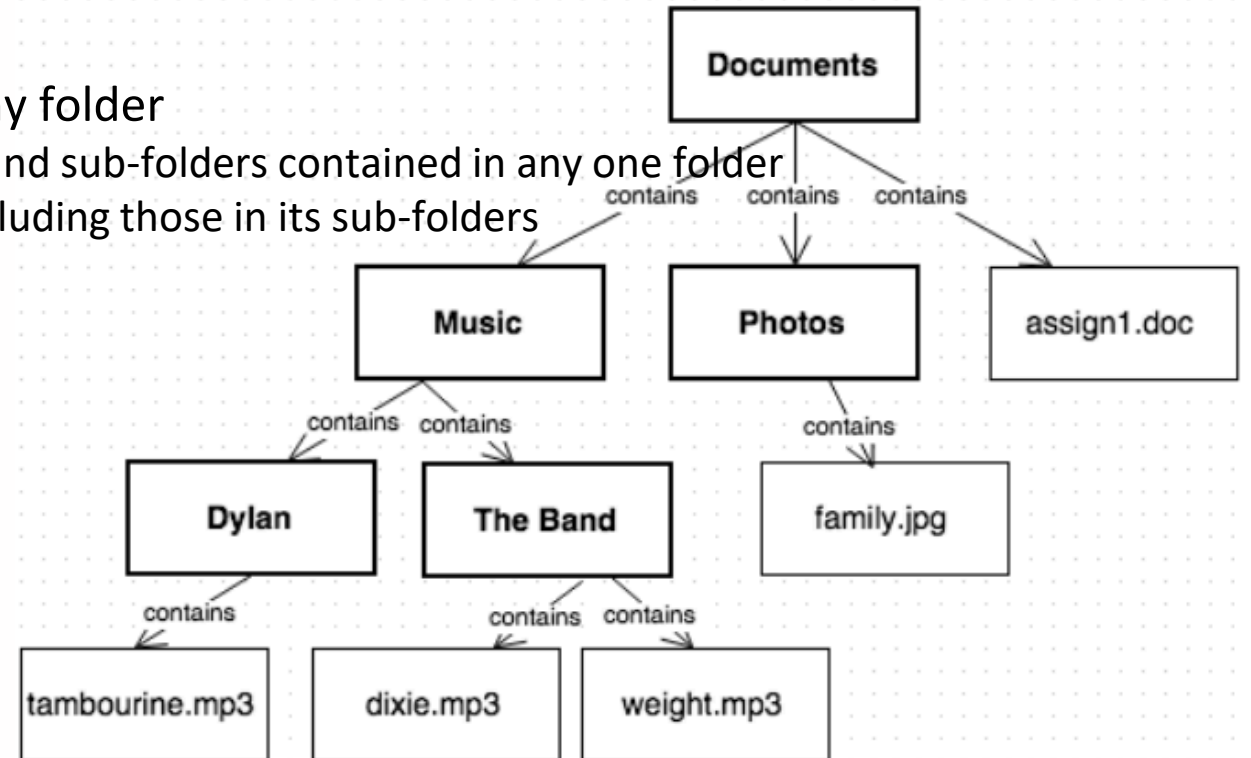


Figure 2.1: an example of a nested folder structure



Where to start?

What information do we have?

1. “The file system should be able to handle folders and files. Each folder can contain files of different types as well as other folders.
2. “request the following from any folder: “size, numFolders numFiles”

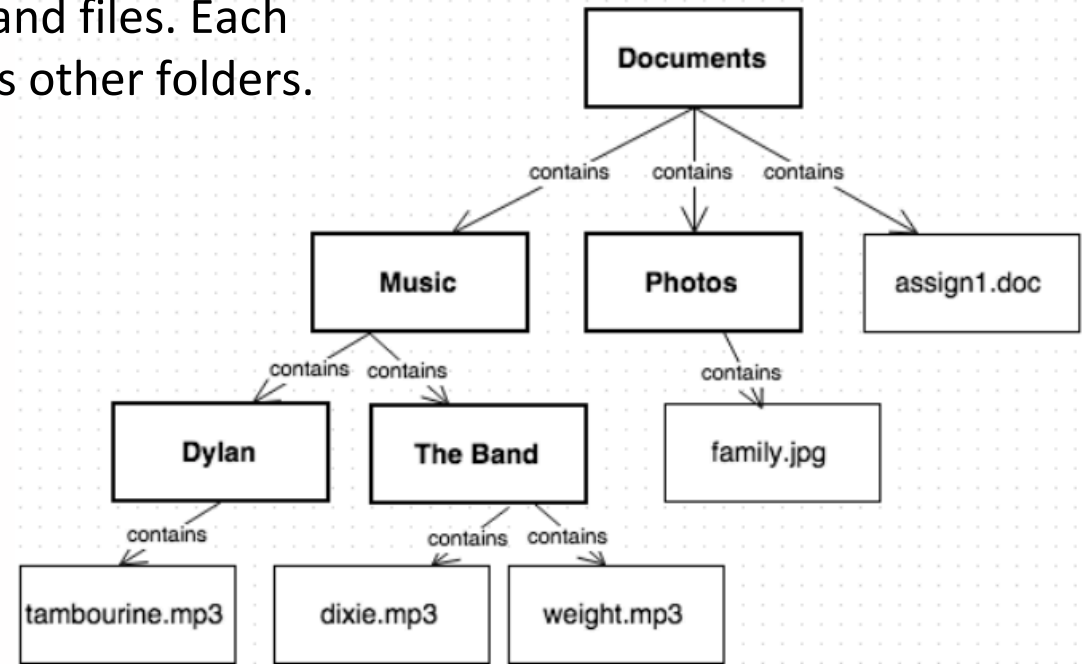


Figure 2.1: an example of a nested folder structure



Preliminary classes

Identify **nouns** in the description above:

- File System
- Folder
- File



Preliminary Associations

- Then identify the relationships
- File System **handles** Folders and Files
- A Folder **can contain** other Files and Folders

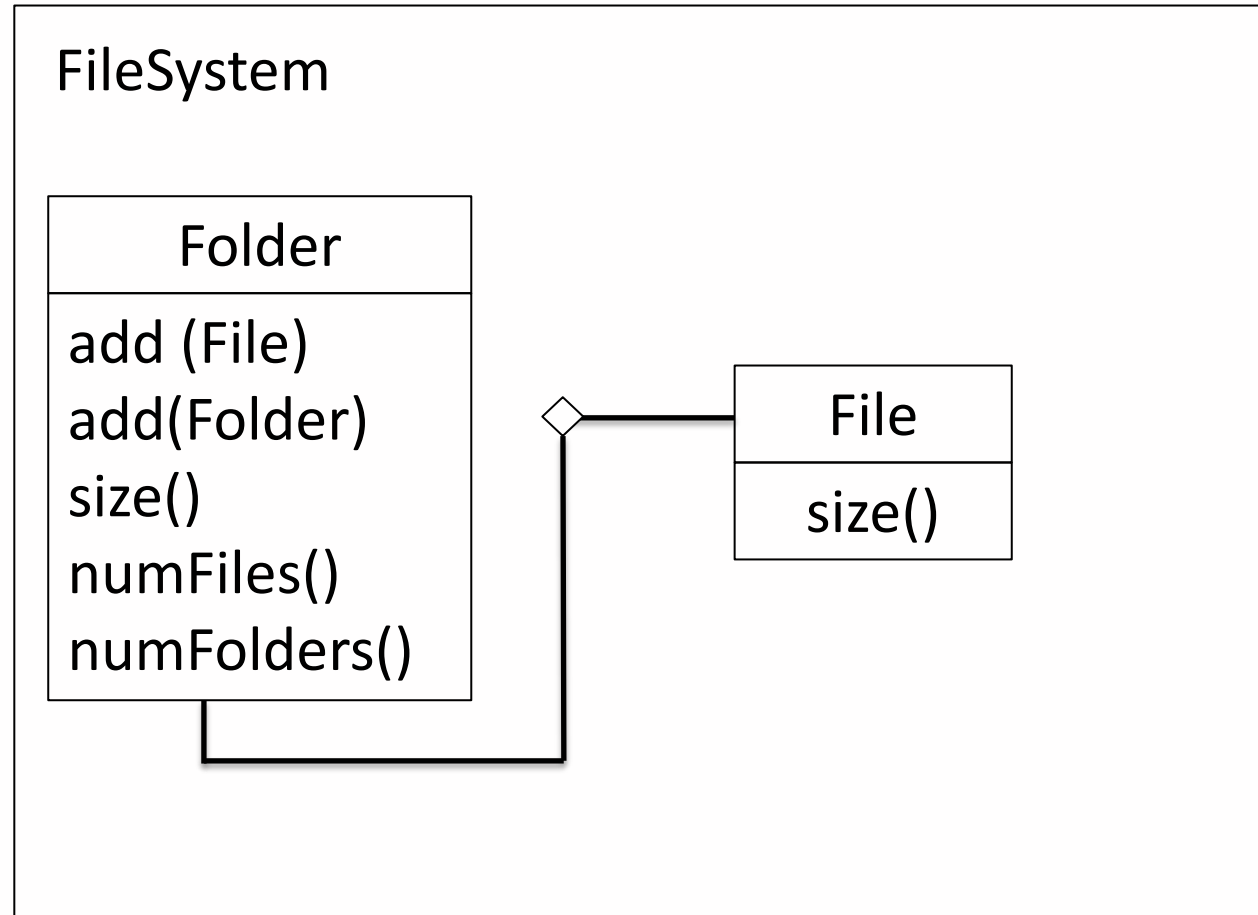


Preliminary Responsibilities

- Identify verbs - however, there is not much to go on
- “You should be able to **request** the following from any folder: size(), numFiles(), numFolders()”
- This suggests that Folder has the responsibility of collecting information from the objects within it
- Since Folder contains other Folders and Files, it must have an **add** method to receive these



Preliminary Class Diagram



The brief we have been given also includes a diagram that illustrates the type of structure that our code should be able to handle
Let's use this example to create a Test method for this scenario
We will then code the stub classes suggested by the the diagram

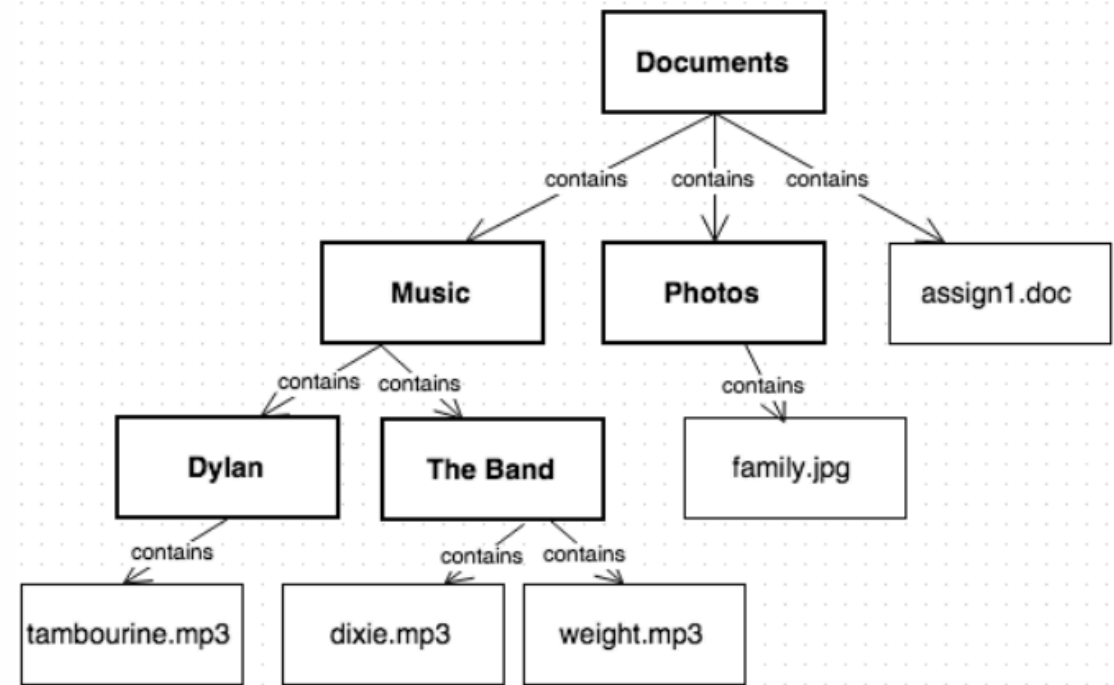


Figure 2.1: an example of a nested folder structure

FileSystem

- Create a class called FileSystem
- Create a main method
- Create a test method – call it fileTest

- Then add the code to fileTest that realises the given example hierarchy
- Create the required classes as you go



FileSystem

For tomorrow, add the remaining test code.

This just means adding a few more Folders and Files to model the example hierarchy

```
/**
 * FileSystem has a main method and is used to
 * simulate different Folder/File scenarios.
 * @author (Conor Hayes)
 * @version (November 9th, 2-17)
 */
public class FileSystem
{
    public static void main(String[] args)
    {
        FileSystem fileSystem = new FileSystem();
        fileSystem.fileTest1();
    }

    public void fileTest1(){

        Folder documents = new Folder("Documents");
        Folder music = new Folder("Music");
        Folder photos = new Folder("Photos");
        documents.add(music);
        documents.add(photos);

        File file = new File("assign1.doc");
        documents.add(file);
        //TODO you should add the remaining code required to complete
        //the example file structure shown in the lecture notes
        // This just requires adding the remaining Folder and File objects
    }
}
```

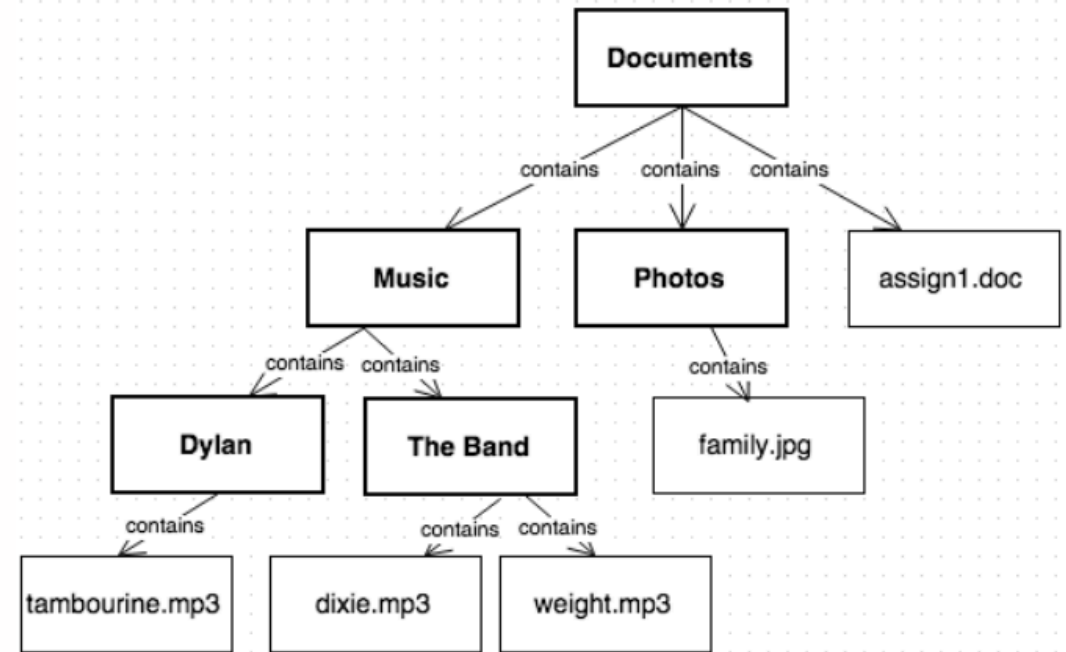


Figure 2.1: an example of a nested folder structure

Applying the composite design pattern

For tomorrow, use what you know from the Composite design pattern to remove the redundant code from this class

```
import java.util.ArrayList;

public class Folder
{
    // instance variables - replace the example below with your own
    private String name;
    private ArrayList<Folder> folders = new ArrayList();
    private ArrayList<File> files = new ArrayList();
    /**
     * Constructor for objects of class Folder
     */
    public Folder(String name)
    {
        // initialise instance variables
        this.name = name;
    }

    public boolean add(Folder folder)
    {
        return folders.add(folder);
    }

    public boolean add(File file)
    {
        return files.add(file);
    }
}
```



Composite: Solution

- The key is an **abstract class** that represents both primitive File elements and their Folder containers.
- The abstract class should represent any common functionality or fields of its sub-classes



Lecture Summary: Composite pattern

- The composite pattern defines how to implement a hierarchical data structure consisting of **primitive** objects and **composite** objects
- Composite objects and primitive objects are both treated in the same way (because they implement the same interface)
- This makes it easy to add new types of components (e.g. new Service class, or more unlikely, a new Canary class)
- All that is required is that these new types of components implement the required interface.





OLLSCOIL NA GAILLIMHE
UNIVERSITY OF GALWAY

CT2106

Object Oriented Programming



Dr. Frank Glavin
Room 404, IT Building
Frank.Glavin@UniversityofGalway.ie
School of Computer Science

University
ofGalway.ie

Lecture Topics

Solving the Folder–File problem

Using the Debugger



OLLSCOIL NA GAILLIMHE
UNIVERSITY OF GALWAY

Folder - File Problem

You are being asked to create a data structure in which a folder contains files and other folders
This requires at the minimum two classes: a Folder (or Directory) class and a File class

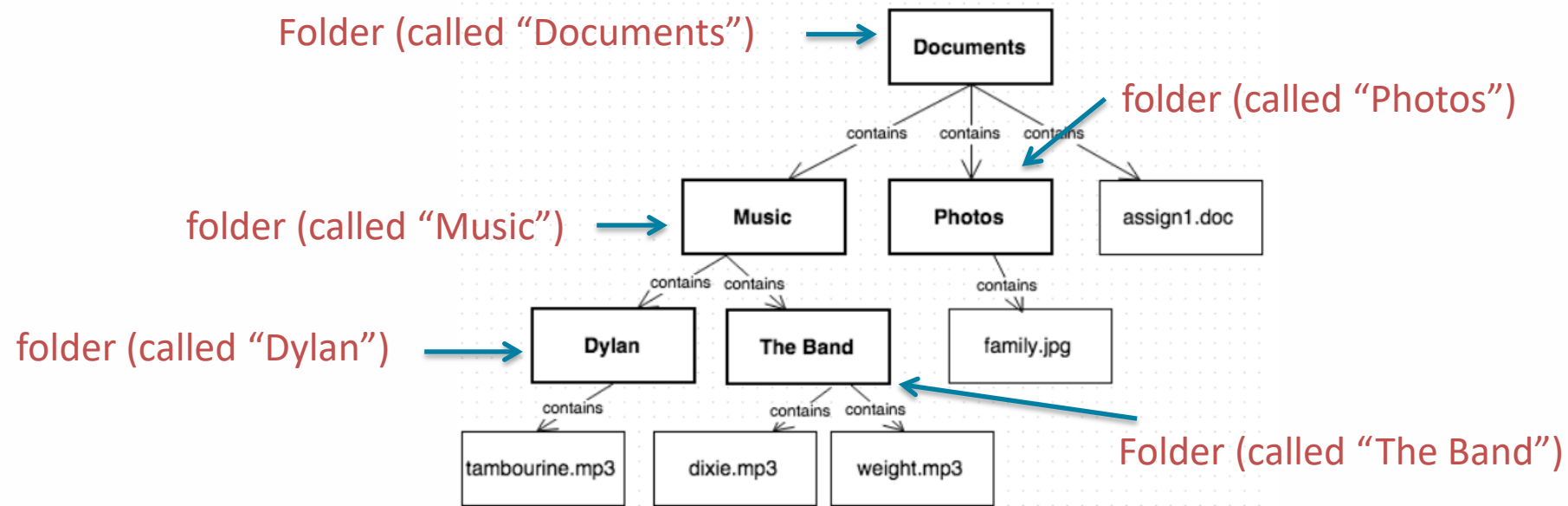


Figure 2.1: an example of a nested folder structure



Potential Mistakes

- Creating Music, Photos and Dylan classes
- What is wrong with this approach?

- Music, Photos and Dylan are three **objects** of the Folder class, with the names “Music”, “Photos” and “Dylan”
- This is a case in which the designer failed to see that Music, Photos and Dylan were each an example of something much more general – a Folder



Yesterday's Requirements

1. Add the remaining test code needed to model the example hierarchy

```
/**
 * Filesystem has a main method and is used to
 * simulate different Folder/File scenarios.
 *
 * @author (Ihsan Ullah)
 * @version (Nov 15)
 */
public class FileSystem
{
    // instance variables - replace the example below with your own
    public static void main(String[] args)
    {
        FileSystem fileSystem = new FileSystem();
        fileSystem.fileTest1();
    }

    public void fileTest1(){
        Folder documents = new Folder("Documents");
        Folder music = new Folder("Music");
        Folder photos = new Folder("Photos");
        documents.add(music);
        documents.add(photos);
        File file = new File("assign1.doc");
        documents.add(file);
        //TODO complete the remaining code so as to complete the file structure shown in the lecture notes
        // It just requires adding the remaining Folder and File objects
    }
}
```

2. Use the composite design pattern to remove the redundancy

```
import java.util.ArrayList;

public class Folder
{
    // instance variables - replace the example below with your own
    private String name;
    private ArrayList<Folder> folders = new ArrayList();
    private ArrayList<File> files = new ArrayList();
    /**
     * Constructor for objects of class Folder
     */
    public Folder(String name)
    {
        // initialise instance variables
        this.name = name;
    }

    public boolean add(Folder folder)
    {
        return folders.add(folder);
    }

    public boolean add(File file)
    {
        return files.add(file);
    }
}
```



1. Adding the remaining test code

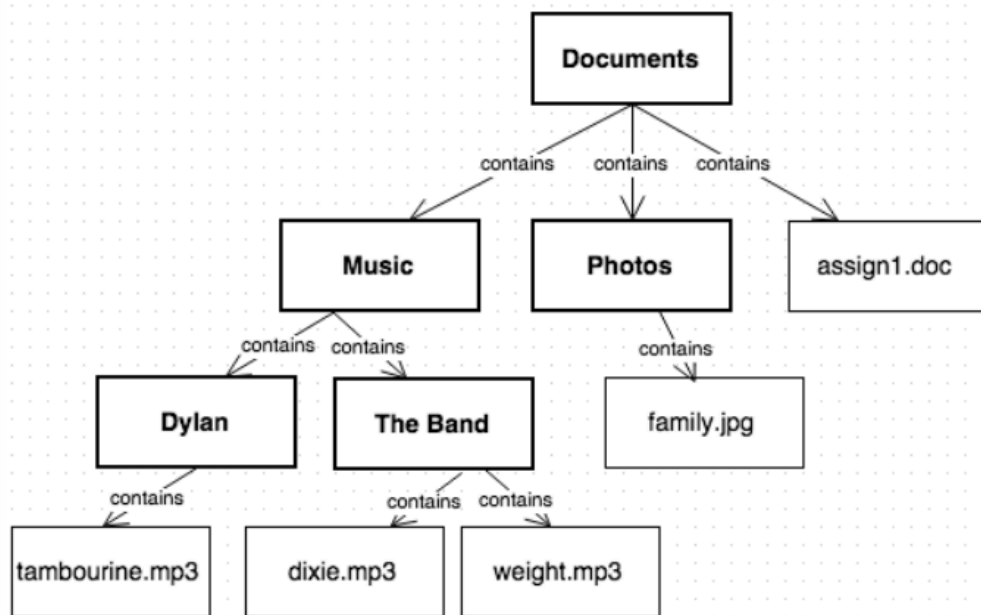


Figure 2.1: an example of a nested folder structure



```
public void fileTest1(){  
    Folder documents = new Folder("Documents");  
    Folder music = new Folder("Music");  
    Folder photos = new Folder("Photos");  
    documents.add(music);  
    documents.add(photos);  
    File file = new File("assign1.doc");  
    documents.add(file);  
    Folder dylan = new Folder("Dylan");  
    music.add(dylan);  
    Folder band = new Folder("The Band");  
    music.add(band);  
    File family = new File("family.jpg");  
    photos.add(family);  
    File tambourine = new File("tambourine.mp3");  
    dylan.add(tambourine);  
    File dixie = new File("dixie.mp3");  
    band.add(dixie);  
    File weight = new File("weight.mp3");  
    band.add(weight);  
}
```



2. Use the *composite* design pattern

- Almost exactly the same approach as with the Assembly-Part solution
- Instead of an interface – I am going to use an abstract class: *AbstractFile*
- Both File and Folder will be types of *AbstractFile*
- *AbstractFile* will define the methods that each of its subclasses should implement



Create an Abstract class

1. Create an abstract class called Abstract File

It should have 4 abstract methods

<code>size();</code>	returns int
<code>getNumFiles();</code>	returns int
<code>getNumFolders();</code>	returns int
<code>find(String name);</code>	returns AbstractFile

2. File and Folder should extend AbstractFile
3. File and Folder should implement all the methods above
4. For now create stub methods – i.e. they simply return default values



AbstractFile

```
public abstract class AbstractFile
{
    // instance variables -
    String name;

    public abstract int size();
    public abstract int getNumFiles();
    public abstract int getNumFolders();
    public abstract AbstractFile find(String name);

    public String getName(){
        return name;
    }
}
```



```

public class Folder extends AbstractFile
{
    //Replace previous ArrayLists with a single
    //ArrayList of AbstractFile references
    private ArrayList<AbstractFile> files = new ArrayList();

    /**
     * Constructor for objects of class Folder
     */
    public Folder(String name)
    {
        super();
        this.name = name;
    }

    // replace previous add methods
    // with a single add(AbstractFile fileObject) method
    public boolean add(AbstractFile fileObject)
    {
        return files.add(fileObject);
    }

    @Override
    public int size()
    {
        //TODO
        return 0;
    }

    @Override
    public int getNumFiles(){
        //TODO
        return 0;
    }

    @Override
    public int getNumFolders(){
        //TODO
        return 0;
    }

    @Override
    public AbstractFile find(String name){
        //TODO
        return null;
    }
}

```

Changes that you need to make to the Folder class

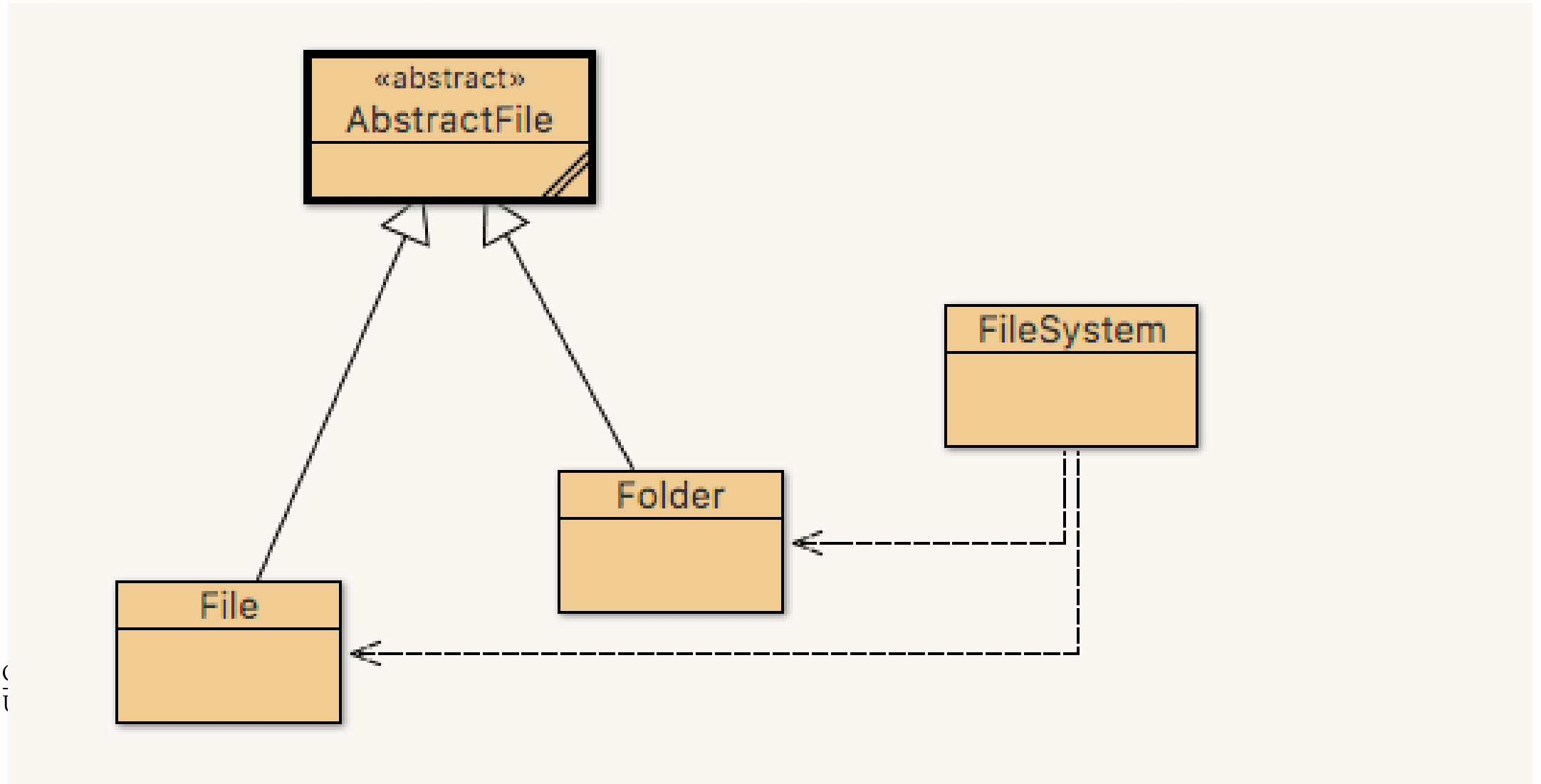



```
public class File extends AbstractFile
{
    private String contents;
    /**
     * Constructor for objects of class File
     */
    public File(String name)
    {
        super();
        this.name = name;
    }
    @Override
    public int size()
    {
        //TODO
        return 0;
    }
    @Override
    public int getNumFiles(){
        //TODO
        return 0;
    }
    @Override
    public int getNumFolders(){
        //TODO
        return 0;
    }
    @Override
    public AbstractFile find(String name){
        //TODO
        return null;
    }
}
```

Changes that you need to make to the Folder class



Revised class diagram



```

public class FileSystem
{
    public static void main(String[] args)
    {
        FileSystem fileSystem = new FileSystem();
        fileSystem.fileTest1();
    }

    public void fileTest1(){

        Folder documents = new Folder("Documents");
        Folder music = new Folder("Music");
        Folder photos = new Folder("Photos");
        documents.add(music);
        documents.add(photos);
        File file = new File("assign1.doc");
        documents.add(file);
        Folder dylan = new Folder("Dylan");
        music.add(dylan);
        Folder band = new Folder("The Band");
        music.add(band);
        File family = new File("family.jpg");
        photos.add(family);
        File tambourine = new File("tambourine.mp3");
        dylan.add(tambourine);
        File dixie = new File("dixie.mp3");
        band.add(dixie);
        File weight = new File("weight.mp3");
        band.add(weight);
    }
}

```

The code compiles
 Now we can start to to implement the
 stub methods and test them in the
 fileTest method across



Test

However, our filetest1 method code is not really **tested** until we can make it pass a **test of some sort**

To do that we should look at each method we are required to create and calculate what each method **should return**

We should evaluate the method based on its **expected output**



Required Methods

Methods

`size()`
`getNumFiles()`
`getNumFolders()`
`find("weight.mp3")`

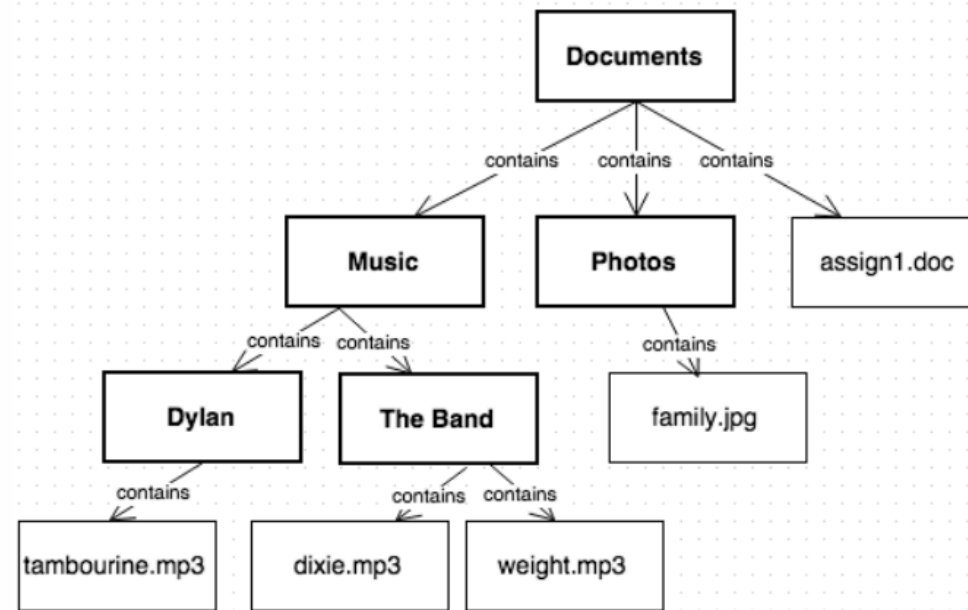


Figure 2.1: an example of a nested folder structure

What are the expected values returned by these methods?



Example test

What is the expected value if we call the *size()* method on the documents folder?

```
int expected = ?  
  
int result = documents.size();  
  
if(result==expected){  
    System.out.println("size() works");  
}else{  
    System.out.println("size() doesn't work");  
}
```

Place this code at the end of the fileTest method



size() method of Folder

- The *size* of a Folder is the sum of the sizes of the Files within the folder.
- This requires adding up the the sizes of all the files within the folder *and its subfolders*
- Same approach to calculate *cost* in the Assembly class
- The *size()* method for Folder is going to look like:

```
@Override
public int size()
{
    int size =0; // size holds the running total
    for (AbstractFile file : files){ // for each AbsFile ref
        size+=file.size(); //call size() and update the running total
    }
    return size; // return the final value
}
```



size() method of File

- In a real world file system, the size of a single file might be the number of bytes on disk
- In our case, we will simplify greatly
- The size of a file will simply the number of *characters* it holds in its **contents** field
- **So lets modify the File class**
 - Add a *contents* field of type String
 - Create the corresponding *getter/setter methods*




```
public class File
{
    // instance variables |
    private String name;
    private String contents;

    /**
     * Constructor for objects of class File
     */
    public File(String name)
    {
        // initialise instance variables
        this.name = name;
    }

    public String getContents(){
        return contents;
    }

    public void setContents(String contents){
        this.contents = contents;
    }
}
```



Contents

This allows us to write (in code pad)

```
File poem = new File("about a cow");  
poem.setContents("How Now, Brown Cow");
```

Now create a *size()* method that returns the number of characters in the file content field



size()

We can use the *length()* method of the String class to return the number of characters in any String

```
@Override
public int size()
{
    if(contents==null){ //contents may not have been set
        return 0;
    }
    return contents.length();
}
```

```
File poem = new File("about a cow");
poem.setContents("How Now, Brown Cow");
int size = poem.size();
size
    18 (int)
```



size() method of Folder

The size() method in Folder adds up the the sizes of all files within the folder *and its subfolders*

```
@Override
public int size()
{
    int size =0; // size holds the running total
    for (AbstractFile file : files){ // for each AbsFile ref
        size+=file.size(); //update the running total
    }
    return size; // return the final value
}
```



Modify the test and run

Now lets write a test for this method

```
String contents1 = "Hey, mister, can you tell me";
String contents2 = "Hey Mr Tambourine Man";
String contents3 = "The night they drove old dixie down";
String contents4 = "fee fi fo fum";

weight.setContents(contents1); // add contents to each File
tambourine.setContents(contents2);
dixie.setContents(contents3);
assign1.setContents(contents4);

//*****test for size()*****
int expected = contents1.length() + contents2.length() +
contents3.length() + contents4.length();
int result = documents.size();

if(result==expected){ // test fro equality
    System.out.println("size() works");
}else{
    System.out.println("size() doesn't work");
}
```



Using the Debugger



Debugger

- Any debugger will have the following core functionalities:
- **Set breakpoints:** set where you want the execution of your program *to pause*
- **Inspect variable values :** inspect the value of variables that are in scope at the breakpoint
- **Step :** tell your program to execute the next line of code. You can inspect the variable values at this point
- **Step into :** tell the debugger to step into a method. You can inspect the values of variables in the method. You can step through lines of code within the method
- **Continue:** tell the debugger to execute the program at normal speed until the next break point or until the end of the program.



Debugger

- This short video on YouTube is also a good tutorial on how to use the debugger
- https://www.youtube.com/watch?v=w_iy0jmMmkA



Summary: Composite Pattern

- The composite pattern defines how to implement a hierarchical data structure consisting of **primitive** objects and **composite** objects
- Composite objects and primitive objects are both treated in the same way (because they implement the *same interface* or extend the *same abstract class*)
- In the example in this lecture, File and Folder are treated the same way – as types of AbstractClass
- This greatly simplifies the code you need to write.





OLLSCOIL NA GAILLIMHE
UNIVERSITY OF GALWAY

CT2106

Object Oriented Programming



Dr. Frank Glavin
Room 404, IT Building
Frank.Glavin@UniversityofGalway.ie
School of Computer Science

University
ofGalway.ie

Today's Topics

- Static Fields
- Exceptions



Card assignment from a previous year

You will find a package called casino containing four classes:

- Card - representing a playing card object
- Deck - representing a deck of playing cards
- Hand - representing a hand of cards (e.g. 5 cards)
- Dealer - a dealer that can shuffle and deal out hands of cards

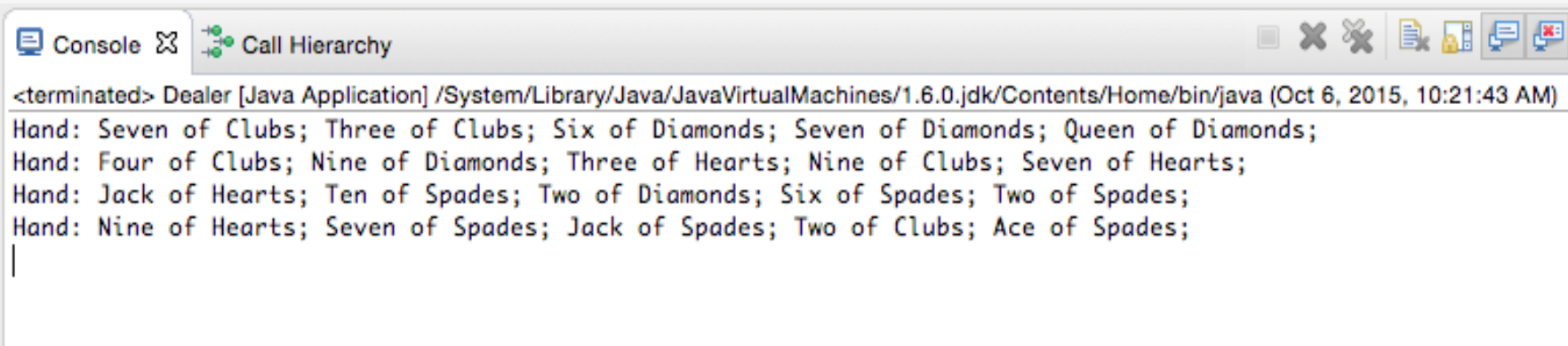
The Dealer class contains the main method.

The programme is called like this:

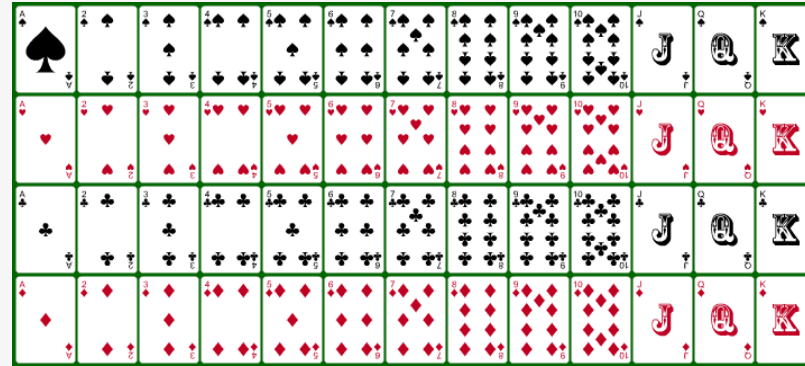
```
java casino.Dealer 5 4
```

This asks the program to deal and print out 4 hands containing 5 playing cards each

It should return output like the following:



```
Console Call Hierarchy  
<terminated> Dealer [Java Application] /System/Library/Java/JavaVirtualMachines/1.6.0.jdk/Contents/Home/bin/java (Oct 6, 2015, 10:21:43 AM)  
Hand: Seven of Clubs; Three of Clubs; Six of Diamonds; Seven of Diamonds; Queen of Diamonds;  
Hand: Four of Clubs; Nine of Diamonds; Three of Hearts; Nine of Clubs; Seven of Hearts;  
Hand: Jack of Hearts; Ten of Spades; Two of Diamonds; Six of Spades; Two of Spades;  
Hand: Nine of Hearts; Seven of Spades; Jack of Spades; Two of Clubs; Ace of Spades;  
|
```



Details

- A card game involves cards of different values
- These are normally gathered together in a Deck
- There are a number of things you might want to do with a deck
 - Shuffle the deck
 - Deal the deck
 - Sort the deck
 - Search for a card



The Card Class

Each Card has a **suit** and a **rank** –represented as instance variables.

suit

Spades	↔	3
Hearts	↔	2
Diamonds	↔	1
Clubs	↔	0

rank

Jack	↔	11
Queen	↔	12
King	↔	13



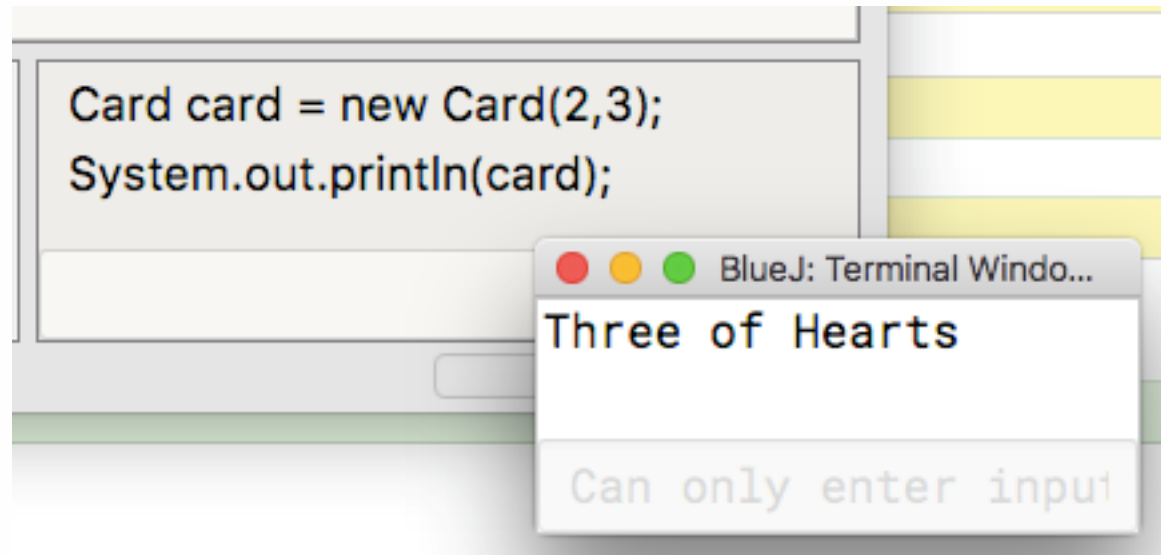
Simple Card Class

```
public class Card {  
  
    private int suit, rank;  
  
    public Card (int s, int r) {  
        this.suit = s; this.rank = r;  
    }  
  
    public int getSuit(){  
        return suit;  
    }  
  
    public int getRank(){  
        return rank;  
    }  
}
```



Card Class

What if you want to be able to print out the value of this Card using the **toString()** method
E.g



Linking suit and rank

We need to link the suit and rank *int* values to the String values representing the Card

Suit: 2 -> "Hearts"

Rank: 3 -> "Three"



You can declare an array of Strings to hold all possible rank values

```
String[] suits = new String[4];
```

And then assign values to the elements:

```
suits[0] = "Clubs";  
suits[1] = "Diamonds";  
suits[2] = "Hearts";  
suits[3] = "Spades";
```

Or you can declare and assign values all in one go

```
String[] suits = {"Clubs", "Diamonds", "Hearts", "Spades"};
```



Card

```
public class Card {  
    private int suit, rank;  
    private String[] suits = {"Clubs", "Diamonds", "Hearts", "Spades"};  
    public Card (int s, int r) {  
        this.suit = s; this.rank = r;  
    }  
    public int getSuit(){  
        return suit;  
    }  
    public int getRank(){  
        return rank;  
    }  
    @Override  
    public String toString(){  
        return suits[suit]; // this returns the suit value but not the rank  
    }  
}
```



rank

- We can do something similar to hold the possible 13 values of the rank of a Card

```
"Ace", "2", "3", "4", "5", "6"
```

```
"7", "8", "9", "10", "Jack", "Queen", "King"
```

```
String[] ranks = {null, "Ace", "2", "3", "4", "5", "6",  
                  "7", "8", "9", "10", "Jack", "Queen", "King"};
```



Q: Why is null the first value in the RANKS array?



```

public class Card {

    private int suit, rank;

    private String[] suits = {"Clubs", "Diamonds", "Hearts", "Spades"};
    private String[] ranks = {null, "Ace", "Two", "Three", "Four", "Five",
                               "Six", "Seven", "Eight", "Nine", "Ten",
                               "Jack", "Queen", "King"};

    public Card (int s, int r) {
        this.suit = s; this.rank = r;
    }

    public int getSuit(){
        return suit;
    }

    public int getRank(){
        return rank;
    }

    @Override
    public String toString(){
        return ranks[rank] + " of " + suits[suit]; //returns rank of suit
    }
}

```



The screenshot shows the BlueJ IDE interface for a project named "Casinov1". On the left, there are buttons for "New Class...", a right-pointing arrow, and "Compile". The main workspace contains a class icon and a class diagram for "Card". Below the workspace, there is a code editor with the following Java code:

```
Card card = new Card(1,4);  
System.out.println(card);
```

A terminal window titled "BlueJ: Terminal Window - Casinov1" is overlaid on the bottom right, displaying the output "Four of Diamonds". At the bottom left of the IDE, the text "card1 : Card" is visible.



Blackboard

- Download the Card Code from Blackboard, Week 11.
- Add it to a project in BlueJ



Introducing static fields

```
private String[] suits = {"Clubs", "Diamonds", "Hearts", "Spades"};  
private String[] ranks = {null, "Ace", "Two", "Three", "Four", "Five",  
                           "Six", "Seven", "Eight", "Nine", "Ten",  
                           "Jack", "Queen", "King"};
```

- Suits and ranks arrays are declared in every object of type Card
- This is wasteful (in terms of memory) and redundant (bad programming practice)
- The suits and ranks values are constant. They never change. They are the same for every Card object
- In situations like this, you should declare these variables to be **static**



Static fields

- Up to now, the instance variables/fields you have used have scope at object level
- A static field is a variable that exists and has scope at **class** level
- Typically, it is used to hold constant, *non-changing* values
- Often, they may be declared public **and final**.
- This means that they can be accessed directly by other classes and objects but *cannot* be changed



Static fields

- Generally, Static variables are capitalised
- Generally declared as **public**
- Very often declared as **final**

- You use them when you want to declare a value/property that is **unchanging or common to all objects of a class**



Static fields

When referring to a static field, use the form
`ClassName.STATIC_VARIABLE_NAME`

E.g

Card.RANKS

Card.SUITS

Math.PI



```

public class Card {

    private int suit, rank;

    public static final String[] SUITS = {"Clubs", "Diamonds", "Hearts", "Spades"};
    public static final String[] RANKS = {null, "Ace", "Two", "Three", "Four", "Five",
        "Six", "Seven", "Eight", "Nine", "Ten",
        "Jack", "Queen", "King"};

    public Card (int s, int r) {
        this.suit = s; this.rank = r;
    }

    public int getSuit(){
        return suit;
    }

    public int getRank(){
        return rank;
    }

    @Override
    public String toString(){
        return Card.RANKS[rank] + " of " + Card.SUITS[suit]; //returns rank of suit
    }
}

```



BlueJ: Casinov1

New Class... [] [] []

→

Compile

Card

card1 : Card

private int suit 2 Inspect

private int rank 3 Get

Show static fields

Class Card

public String[] SUITS Inspect

public String[] RANKS Get

Close

```

class Card {
    private int suit, rank;
    static final String[] SUITS = {"Cl", "Sp", "He", "Dc"};
    static final String[] RANKS = {"Ace", "Two", "Three", "Four", "Five", "Six", "Seven", "Eight", "Nine", "Ten", "Jack", "Queen", "King"};
    Card(int s, int r) {
        suit = s; rank = r;
    }
    int getSuit() {
        return suit;
    }
}

```

card1 : Card

card1 : Card



Exception Handling



OLLSCOIL NA GAILLIMHE
UNIVERSITY OF GALWAY

Card Class

Our Card class has a significant weakness

```
public class Card {  
  
    private int suit, rank;  
  
    public static final String[] SUITS = {"Clubs", "Diamonds", "Hearts", "Spades"};  
    public static final String[] RANKS = {null, "Ace", "Two", "Three", "Four", "Five",  
                                           "Six", "Seven", "Eight", "Nine", "Ten",  
                                           "Jack", "Queen", "King"};  
  
    public Card (int s, int r) {  
        this.suit = s; this.rank = r;  
    }  
  
    public int getSuit(){  
        return suit;  
    }  
  
    public int getRank(){  
        return rank;  
    }  
  
    @Override  
    public String toString(){  
        return Card.RANKS[rank] + " of " + Card.SUITS[suit]; //returns rank of suit  
    }  
}
```



Handling invalid values

```
Card card = new Card(24,17);  
Card card2 = new Card(99,104);  
System.out.println(card);  
Exception: java.lang.ArrayIndexOutOfBoundsException (17)
```

```
java.lang.ArrayIndexOutOfBoundsException: 17  
at Card.toString(Card.java:26)  
at java.lang.String.valueOf(String.java:2994)  
at java.io.PrintStream.println(PrintStream.java:821)
```

- It allows us to create Card objects with invalid Card values.
- The error will only be detected later in the program



An Exception

```
java.lang.ArrayIndexOutOfBoundsException: 17
    at Card.toString(Card.java:26)
    at java.lang.String.valueOf(String.java:2994)
    at java.io.PrintStream.println(PrintStream.java:821)
```

- The error message above is from the Java Runtime Environment (JRE)
- It tells use that an Exception was generated and was not handled
- This has caused the program to crash



What is an Exception?

- An exception is an “*exceptional event*” – one that may lead to a serious error in your program if not handled appropriately.
- An exception is generated only when the program runs – hence it is known as a **runtime error**
- Very often, the error (and the exception generated) occurs **when the program is asked to do something that is impossible for it to do**
- In Java, each exception is represented by an **Exception object**



Programming for Exceptions

- As the programmer, it your responsibility to anticipate the situations in which your program will fail
- You have to write code to manage any **exceptional events** that may occur within your program
- In our example, an exceptional event is when a user tries to get our program to instantiate an invalid card

```
Card card1 = new Card(23,21);
```

- If this card object gets into say, a poker program, it will wreak havoc, as all other objects will expect Card objects with valid suit and rank values



Checking valid input for a Card

- The key question is how to programmatically handle the situation when invalid input is entered.
- In the case of the Card, we might write the following in the constructor:

```
public Card (int suit, int rank) {  
    if(suit<0 || suit> Card.SUITS.length-1){  
        System.out.printf("Incorrect suit value %d ",suit);  
    }  
  
    if(rank<1 || rank> Card.RANKS.length-1){  
        System.out.printf("Incorrect rank value %d ",rank);  
    }  
  
    this.suit = suit;  this.rank = rank;  
}
```



Weak approach

- It prints out a warning message only
- The invalid Card object is still created

```
Card card = new Card(-9, 47);
card
<object reference> (Card)
```

BlueJ: Terminal Window - Casinov1

```
Incorrect suit value -9 Incorrect rank value 47
```



Detect error-> Throw an Exception

- We want an approach that **prevents an invalid object being created**
- Java has the concept of an Exception object that can be created to stop a program going any further
- When a program generates an Exception object it is said to **throw an Exception**
- When an Exception is thrown, the program must have code in place to **catch it**
- **If not, the program terminates**



Throwing an Exception

This involves

1. Detecting an error
2. Creating an Exception object
3. Passing the Exception object to The Java Runtime Environment (JRE) Exception Handling Procedures.
This also means the execution of the method does not complete
4. The JRE then looks for part of your program to take responsibility for this error.
5. In other words, your program should also have code ready to **catch** the error



Card throws `IllegalArgumentException`

In our case, we can make the Card throw an Exception - an **`IllegalArgumentException`**

```
public class IllegalArgumentException  
    extends RuntimeException
```

Thrown to indicate that a method has been passed an illegal or inappropriate argument.



throws

When you want a method to throw an Exception you add **throws** and the Exception type to the method signature

```
public Card (int suit, int rank) throws IllegalArgumentException {
```

This tells any code that wants to call the constructor method that it may throw an **IllegalArgumentException**

It will be up to the calling code to handle that exception if it is thrown



throws

- The Card constructor has to define conditions which will cause it to throw an Exception.
- These are the same conditions that caused it to issue a weak warning
- Instead now, it generates and **throws** a new Exception object
- To throw an Exception you use the **throw** keyword



Revised Card constructor

- When an Exception is thrown, execution of the method stops
- As this is a *constructor* method, this means that the (invalid) Card object is not created

```
public Card (int suit, int rank) throws IllegalArgumentException {  
    if(suit<0 || suit> Card.SUITS.length-1){  
        throw new IllegalArgumentException("Incorrect suit value " +suit);  
    }  
  
    if(rank<1 || rank> Card.RANKS.length-1){  
        throw new IllegalArgumentException("Incorrect rank value " + rank);  
    }  
  
    this.suit = suit;  this.rank = rank;  
}
```



Testing out code

- Now when we try to create a Card with invalid values, we will fail.
- An exception is thrown.
- The card variable below is not assigned to a Card object

```
Card card = new Card(-34, 78);  
    Exception: java.lang.IllegalArgumentException (Incorrect suit value -34)  
card  
    Error: cannot find symbol - variable card
```



throwing and catching

- If your method **throws** an exception
- Then you **must** also have code in place to **catch** and **handle** the exception



Graceful recovery

- If an exception is not caught, the JRE will terminate the program
- This is a drastic step
- In most cases, you will want your program to recover (gracefully) from an exception and carry on
- This involves **catching** the Exception that has been generated



Example of program termination

- If you run the following code, the **uncaught exception** will terminate the program at line 18

```
14 public static void main (String[] args)
15 {
16     Card card1 = new Card(0,1); //valid card
17
18     Card card2 = new Card(0,-1); //invalid card
19
20     Card card3 = new Card(0,2); //valid card
21
22     System.out.println(card1);
23     System.out.println(card2);
24     System.out.println(card3);
25
26 }
```

- Nothing after line 18 will execute

```
java.lang.IllegalArgumentException: Incorrect rank value -1
at Card.<init>(Card.java:19)
at CardTest.main(CardTest.java:18)
```



Try/catch

- If you want the program to recover from the Exception, you have to catch and handle it
- This means using a try/catch expression
- **Try:** try to execute this piece of code. If it executes without throwing an exception. Fine. There is no need to for **the catch** clause to be executed
- **Catch:** if an exception has been thrown then execute this piece of recovery code to **handle** the Exception (very often just an error message)



General format of try/catch block

Meaning:

1. Try to call this method (which may throw an Exception)
2. If it throws an exception object, catch it! (the exception will go no further)
3. Then handle the exception this way
4. Carry on to the next line of execution (as normal)

```
try{  
    // call the code that may throw an Exception  
}catch{//TheExceptionClass thevariable}{  
    // How you want to handle the error  
}
```



Revised Example

- Each call to the Card constructor is wrapped in a try/catch block
- If an Exception is thrown, it will be caught and handled
- This allows the program to execute until the end.

```
public static void main (String[] args)
{
    Card card1 = null;
    Card card2 = null;
    Card card3 = null;

    try{
        card1 = new Card(0,1); //valid card
    }catch(IllegalArgumentException e){
        System.out.println(e.getMessage());
    }

    try{
        card1 = new Card(0,-11); //invalid card
    }catch(IllegalArgumentException e){
        System.out.println(e.getMessage());
    }

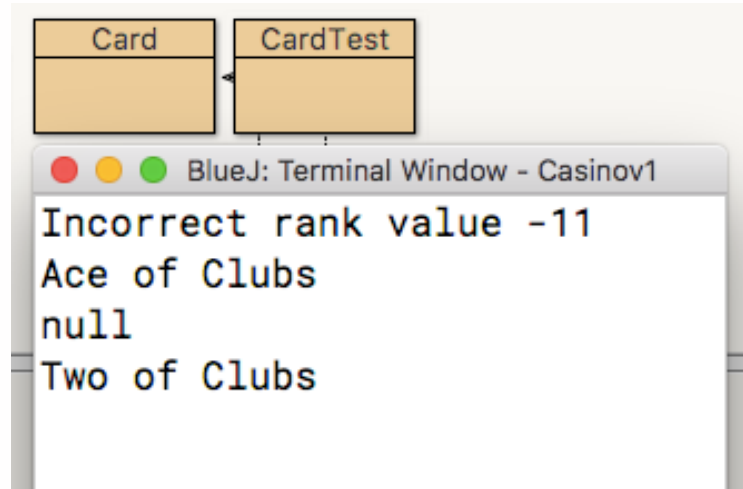
    try{
        card3 = new Card(0,2); //valid card
    }catch(IllegalArgumentException e){
        System.out.println(e.getMessage());
    }

    System.out.println(card1);
    System.out.println(card2);
    System.out.println(card3);
}
```



Graceful recovery

- Now when we run the program we get an error message caused by the attempt to create the invalid second Card



- The invalid second card is not created
- **The program can continue** on to create the third card ("Two of Clubs")
- It then prints out the values of the card1, card2 and card3 variables
- (card2 is pointing to null, because the second invalid card was not created)



Some common unchecked Exceptions

Name	Description
NullPointerException	Thrown when attempting to access an object with a reference variable whose current value is null
ArrayIndexOutOfBoundsException	Thrown when attempting to access an array with an invalid index value (either negative or beyond the length of the array)
IllegalArgumentException.	Thrown when a method receives an argument formatted differently than the method expects.
IllegalStateException	Thrown when the state of the environment doesn't match the operation being attempted, e.g., using a Scanner that's been closed.
NumberFormatException	Thrown when a method that converts a String to a number receives a String that it cannot convert.
ArithmeticException	Arithmetic error, such as divide-by-zero.



Wrapping up

- A static field is a variable that exists and has scope at **class** level
- You use them when you want to declare a value/property that is common to all objects of a class
- You can anticipate when errors may be generated by your program and write exceptions throwing code to cover these events
- You also have to write code to catch and handle **exceptions** that may occur within your program





OLLSCOIL NA GAILLIMHE
UNIVERSITY OF GALWAY

CT2106

Object Oriented Programming



Dr. Frank Glavin
Room 404, IT Building
Frank.Glavin@UniversityofGalway.ie
School of Computer Science

University
ofGalway.ie

Today's Lecture

Using the Comparable Interface

Sorting

Testing



OLLSCOIL NA GAILLIMHE
UNIVERSITY OF GALWAY

Back to the Card assignment

You will find a package called casino containing four classes:

- Card - representing a playing card object
- Deck - representing a deck of playing cards
- Hand - representing a hand of cards (e.g. 5 cards)
- Dealer - a dealer that can shuffle and deal out hands of cards

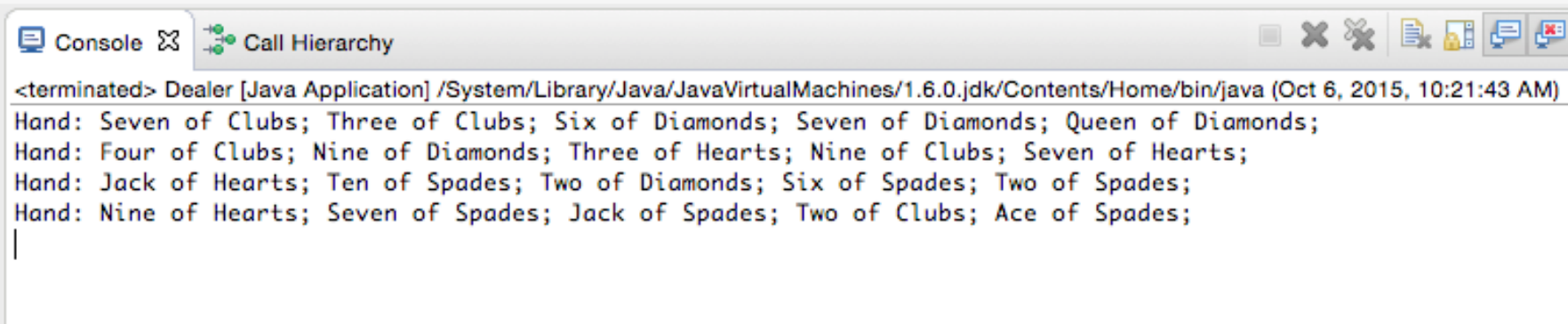
The Dealer class contains the main method.

The programme is called like this:

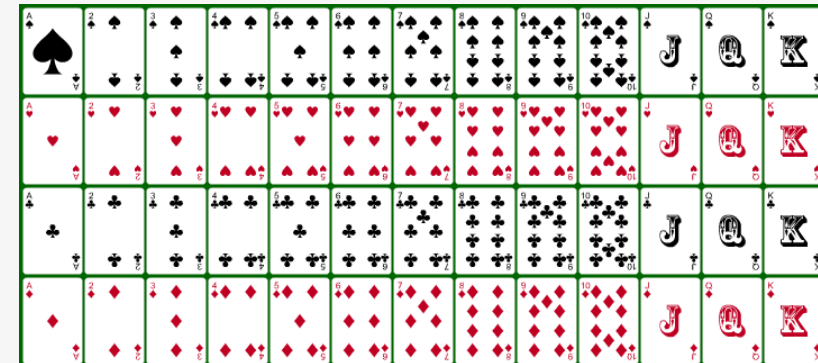
```
java casino.Dealer 5 4
```

This asks the program to deal and print out 4 hands containing 5 playing cards each

It should return output like the following:



```
<terminated> Dealer [Java Application] /System/Library/Java/JavaVirtualMachines/1.6.0.jdk/Contents/Home/bin/java (Oct 6, 2015, 10:21:43 AM)
Hand: Seven of Clubs; Three of Clubs; Six of Diamonds; Seven of Diamonds; Queen of Diamonds;
Hand: Four of Clubs; Nine of Diamonds; Three of Hearts; Nine of Clubs; Seven of Hearts;
Hand: Jack of Hearts; Ten of Spades; Two of Diamonds; Six of Spades; Two of Spades;
Hand: Nine of Hearts; Seven of Spades; Jack of Spades; Two of Clubs; Ace of Spades;
|
```



Card Game

A card game involves cards of different values

These are normally gathered together in a Deck

There are a number of things you might want to do with a deck

- Shuffle the deck

- Deal the deck

- Sort the deck

- Search for a card



```

public class Card {

    private int suit, rank;

    public static final String[] SUITS = {"Clubs", "Diamonds", "Hearts", "Spades"};
    public static final String[] RANKS = {null, "Ace", "Two", "Three", "Four", "Five",
        "Six", "Seven", "Eight", "Nine", "Ten",
        "Jack", "Queen", "King"};

    public Card (int suit, int rank) throws IllegalArgumentException {

        if(suit<0 || suit> Card.SUITS.length-1){
            throw new IllegalArgumentException("Incorrect suit value " +suit);
        }

        if(rank<1 || rank> Card.RANKS.length-1){
            throw new IllegalArgumentException("Incorrect rank value " + rank);
        }

        this.suit = suit; this.rank = rank;
    }

    public int getSuit(){
        return suit;
    }

    public int getRank(){
        return rank;
    }

    @Override
    public String toString(){
        return Card.RANKS[rank] + " of " + Card.SUITS[suit]; //returns rank of suit
    }
}

```



equals ()

Recall that every object inherits `equals` method from `java.lang.Object`
Two cards are equal if they have **the same suit and the same rank**



Quiz: equals() method for Card

```
@Override
public boolean equals(a object){
    if(object==null){
        return b ;
    }

    if (object c Card){
        Card card = (Card) object;

        if(suit==card.getSuit() d rank==card.getRank()){
            return true;
        }
    }

    return e ;
}
```



compareTo

Equals is a very useful method

However, when **searching or sorting**, it is important to know whether one object has a greater/less value than another

With primitive values, it is trivial to understand if one number is greater/less than another.

E.g. $5 > 4$; $0.1 > -0.1$;

How do we decide if one Card is greater/less than other?



Natural Ordering

When deciding on whether one object is greater or less than another, we refer to the **natural ordering** of the objects' class

Natural ordering is the ordering imposed on an object when its class implements the **Comparable** Interface

In Google look-up , *“Java Comparable Interface”*



Comparable<T>

```
public interface Comparable<T>
```

This interface imposes a total ordering on the objects of each class that implements it. This ordering is referred to as the class's *natural ordering*, and the class's `compareTo` method is referred to as its *natural comparison method*.

Lists (and arrays) of objects that implement this interface can be sorted automatically by `Collections.sort` (and `Arrays.sort`). Objects that implement this interface can be used as keys in a sorted map or as elements in a sorted set, without the need to specify a comparator.



Comparable<T> interfaces

Like most interfaces, very lightweight

Has one method: `compareTo`

All classes that implement Comparable, must also provide a concrete implementation of `compareTo`



compareTo(T o)

```
int compareTo(T o)
```

Parameters:

`o` - the object to be compared.

Returns:

a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.

Throws:

`NullPointerException` - if the specified object is null

`ClassCastException` - if the specified object's type prevents it from being compared to this object.



Interface Comparable<T>

The <T> in Comparable<T> means that we *can* specify in advance the type of the object that should be compared

In other words, unlike the equals method which has a generic Object parameter, we can specify the input type for the *compareTo* method



Objective: make the Card class sortable and searchable
Create a Deck of Cards that can be shuffled and searched



implements Comparable

Modify the Class definition of Card to implement Comparable

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

The <Card> tells Java that you plan to compare Card objects only

To get this to compile you have to implement the **compareTo** method

```
@Override  
public int compareTo(Card card){  
    //TODO - Stub implementation  
    return 0;  
}
```



What is the natural ordering of a set of Cards?

The suits are generally ordered in increasing value as follows

clubs, diamonds, hearts, spades

The rank goes is ordered in increasing value

Ace, 2, 3, 4, 5, 6, 7, 8, 9, 10, Jack, Queen, King

These orderings are reflected by the arrays we have already defined

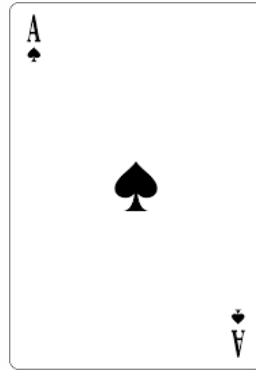
```
SUITS = {"Clubs", "Diamonds", "Hearts", "Spades"};  
RANKS = {null, "Ace", "Two", "Three", "Four", "Five",  
         "Six", "Seven", "Eight", "Nine", "Ten",  
         "Jack", "Queen", "King"};
```



What is the natural ordering of a set of Cards?

The suit value produces the *primary* ordering

Card(3,1)



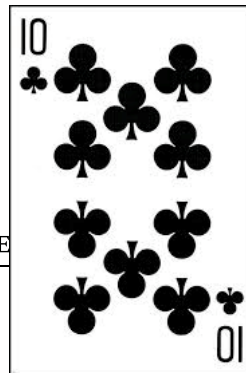
is always
greater than



Card(2,1)

The rank value produces the *secondary* ordering

Card(0,10)



is always
greater than

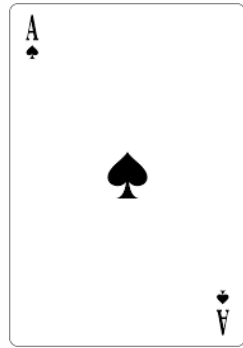


Card(0,9)



OLLSCOIL NA GAILLIMHÉIR
UNIVERSITY OF GALWAY

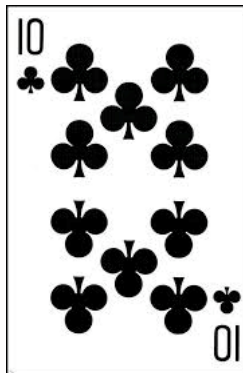
How should `compareTo` behave?



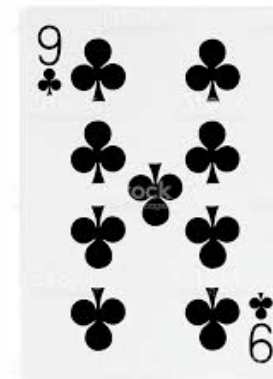
`compareTo`



= 1



`compareTo`



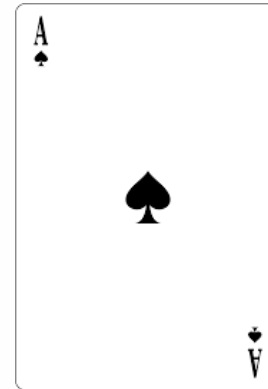
= 1



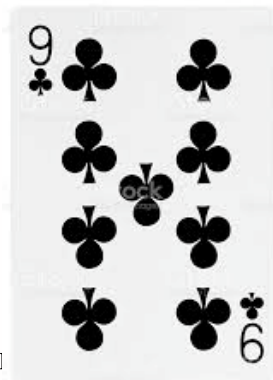
How should `compareTo` behave?



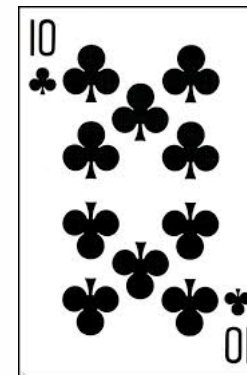
`compareTo`



`= -1`



`compareTo`



`= -1`



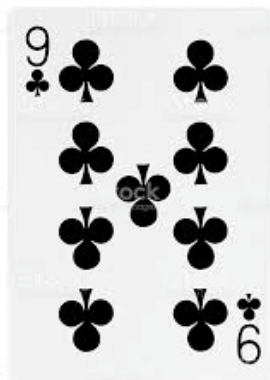
How should `compareTo` behave?



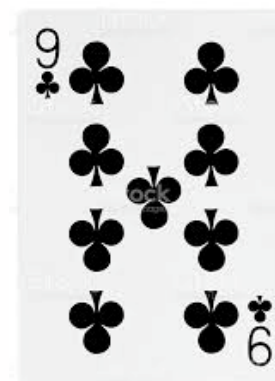
`compareTo`



`= 0`



`compareTo`



`= 0`



Card.compareTo

The method first checks for equality
Then checks if the card is in a higher or lower suit
Then it checks it's rank

```
@Override
public int compareTo(Card card){

    // if this card is equal to card return 0
    // if this suit value is greater than card's suit value return 1
    // if this suit value is less than card's suit value return -1
    // if this rank is greater than card's rank return 1
    // otherwise return -1

}
```



compareTo

The method first checks for equality

Then checks if the card is in a higher or lower suit

Then it checks it's rank

```
@Override
public int compareTo(Card card){
    if(this.equals(card)) return 0; // if equal

    if(this.suit > card.getSuit()) return 1; // if this suit is greater
    if(this.suit < card.getSuit()) return -1; // if this suit is less

    //otherwise the suits are equal

    if(this.rank > card.getRank()) return 1; // if the rank is greater

    return -1; // only possible other option i
}
```



assert

Use **assert** to declare a statement that **must be true**

If it is not true, your programme will throw an AssertionError Exception

You can use the Assert statement as a quick way to test for expected output

```
assert(2==2); // will always be true  
assert(true==false) // will always be false
```



Quick Test

```
public void testCompareTo(){
    Card card1 = new Card(1,2);
    Card card2 = new Card(1,2);

    int result = card1.compareTo(card2);
    assert(result==0); // assert = this must be true

    Card card3 = new Card(2,3);
    Card card4 = new Card(1,2);

    result = card3.compareTo(card4);
    assert(result==1); // assert = this must be true

    result = card4.compareTo(card3);

    assert(result==-1); // assert = this must be true
}
```



If you run this code and it produces no Exception then the assert statements were all true – and your code passed the test

Download the code uploaded after this lecture to test it yourself



A Deck of Cards

We will create a new class called Deck to hold the Card objects
When we create a Deck object, it should immediately populate itself
with 52 card objects
We also want methods to **sort the Cards** and to **search for a Card**



Deck Class

Function: to store cards and to perform any methods to do with *shuffling* and *sorting* and *searching*

What data structure will it use to store the Card objects?



Deck Class

Function: to store cards and to perform any methods to do with sorting and searching

Instance variable is an array of references to Card objects

```
public class Deck
{
    // instance variables
    private Card[] cards = new Card[52];
    ...
}
```



Deck()

Constructor populates the Deck with Card objects

Outer loop enumerates the suits from 0 to 3.

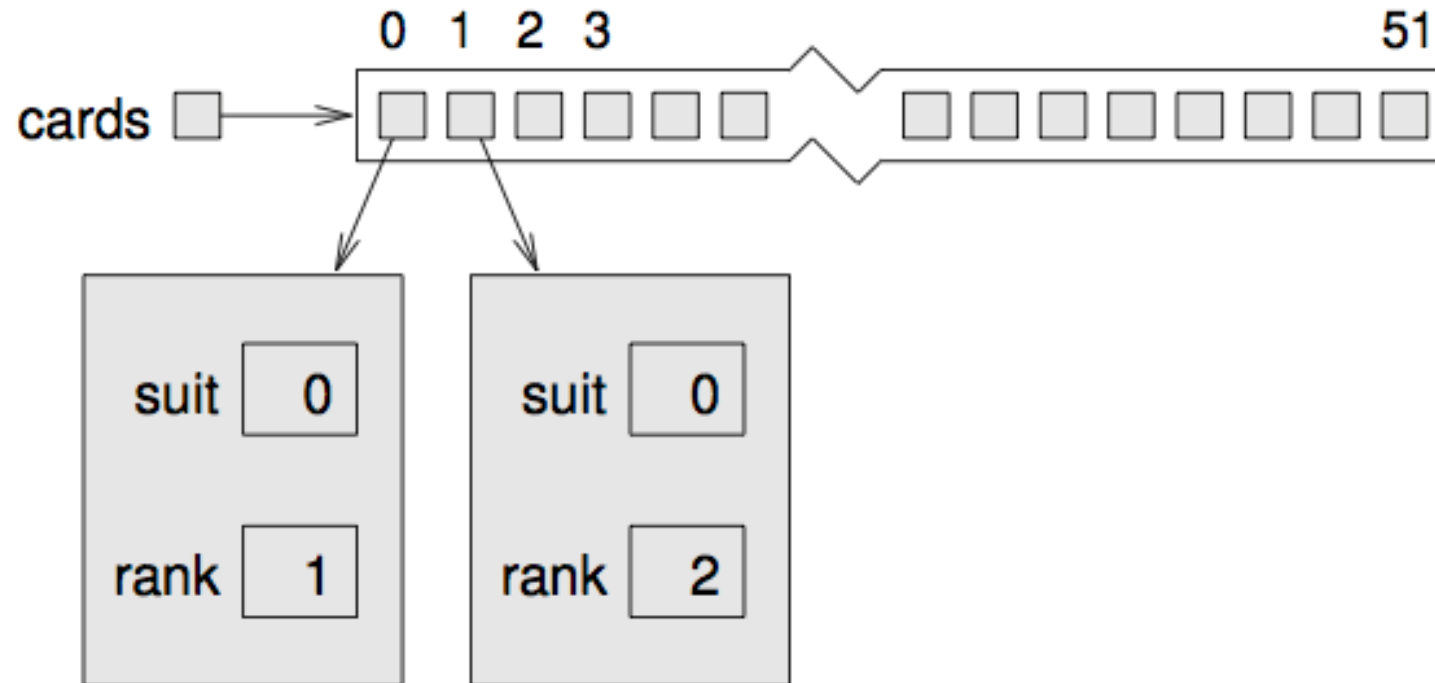
Inner loop enumerates the ranks from 1 to 13.

```
/**
 * Constructor for objects of class Deck
 */
public Deck()
{
    // this code creates 52 unique Cards
    int index = 0;
    for(int i =0 ; i< Card.SUITS.length; i++){ // for each suit value
        for(int j =1 ; j< Card.RANKS.length; j++){ // for each rank value
            cards[index] = new Card(i,j); // add a new Card
            index++; // increase the index by 1
        }
    }
}
```



Card Array

Cards Array now contains 52 Card objects



Sorting

We are going to create an instance method called `sort` belonging to the `Deck` class
It should sort the `Cards` into the order in which they were created by the `Deck`



Arrays.sort

We will make use of the the sort method from the `java.util.Arrays` class

Look up `java.util.Arrays` on Google



```
public class Arrays  
extends Object
```

This class contains various methods for manipulating arrays (such as sorting and searching). This class also contains a static factory that allows arrays to be viewed as lists.

The methods in this class all throw a `NullPointerException`, if the specified array reference is null, except where noted.



sort

```
public static void sort(Object[] a)
```

Sorts the specified array of objects into ascending order, according to the natural ordering of its elements. All elements in the array must implement the Comparable interface. Furthermore, all elements in the array must be *mutually comparable* (that is, `e1.compareTo(e2)` must not throw a `ClassCastException` for any elements `e1` and `e2` in the array).

This sort is guaranteed to be *stable*: equal elements will not be reordered as a result of the sort.



sort

With the Arrays class, creating a sort method for the array of Cards is easy

```
public void sort()  
{  
    Arrays.sort(cards);  
}
```

That's all there is to it.

Remember to put **import java.util.Arrays** at the top of the class



```
import java.util.Arrays;
```

```
public class Deck  
{
```

```
    // instance variables  
    private Card[] cards = new Card[52];
```

```
    /**  
     * Constructor for objects of class Deck  
     */
```

```
    public Deck()  
    {
```

```
        // this code creates 52 unique Cards
```

```
        int index = 0;
```

```
        for(int i =0 ; i< Card.SUITS.length; i++){ // for each suit value
```

```
            for(int j =1 ; j< Card.RANKS.length; j++){ // for each rank value
```

```
                cards[index] = new Card(i,j); // add a new Card
```

```
                index++; // increase the index by 1
```

```
            }
```

```
        }
```

```
    }
```

```
    public void sort()  
    {
```

```
        Arrays.sort(cards);
```

```
    }
```

```
}
```



sort() method in the Deck class

observation: As far as the `Arrays.sort` method is concerned it is sorting an `Array of Comparable` objects, not `Card` objects

The `Arrays.sort` method will only ever call the **compareTo** method of the `Card` object

```
public void sort()
{
    Arrays.sort(cards);
}
```



How do we test the sort method?

Define an **equals** method for Deck

If two Decks have the **same cards, in the same order then they are equal**

Test approach

- Create two decks

- Test if they are equal

- Shuffle one Deck

- Test that the Decks are no longer equal

- Sort the shuffled Deck (with new sort method)

- Test if both decks are equal again



How do we test the sort method?

Define an **equals** method for Deck

If two Decks have the **same cards, in the same order then they are equal**

```
@Override
public boolean equals(Object object){
    if (object == null){
        return false;
    }
    if(object instanceof Deck){
        Deck deck = (Deck) object;
        for(int i = 0; i< cards.length; i++){
            if(!getCard(i).equals(deck.getCard(i))){ //
                return false;// the decks are not equal
            }
        }
    }
    return true;
}
```



How do we test the sort method?

Define a shuffle method for Deck

Many ways to do this

The code below randomly shuffles the array of cards according to the *Fisher Yates algorithm*

```
//This is an implementation of the Fisher Yates Shuffle
public void shuffle(){
    for(int i = cards.length-1; i>0; i--){
        int j = (int)(Math.random() * i+1);
        Card temp = cards[i];
        cards[i] = cards[j]; // exchanging the card at i and j
        cards[j] = temp;
    }
}
```



Test Code

```
public static void main(String[] args)
{
    Deck deck1 = new Deck();
    Deck deck2 = new Deck();

    assert(deck1.equals(deck2)); // should be equal

    deck1.shuffle();// randomly shuffles the deck

    assert(!deck1.equals(deck2)); // both decks should not be equal

    deck1.sort(); // should sort the deck back to its original order

    assert(deck1.equals(deck2)); // should be equal again
}
```



Testing

If this test code runs without throwing an Exception then the assert methods were true
And the code passed the test

Run the code yourself and verify that no AssertionError Exception is thrown
Comment out the deck1.sort() method in the test code.
Verify that an AssertionError Exception is now thrown



Lecture wrap up

- This lecture we looked at using the Comparable interface
- We defined the compareTo method for a Card object
- We then used the java.util.Arrays.sort method to sort a Deck of Cards
- As with any method we design we devised a test to evaluate if the method works
- A handy way of evaluating whether an expected value occurs is to use the `assert` function
- If the assert fails, the program throws an AssertionError alerting you to the fact that your code has not produced expected output

