Week 1 - Friday

COMP 2100

Last time

- What did we talk about last time?
- Basic programming model
- Other Java stuff
  - References
  - Static
  - Inner classes

Questions?

Assignment 1

Project 1

Exceptions

Exceptions

- Java handles errors with **exceptions**
- Code that goes wrong **throws** an exception
- The exception propagates back up through the call stack until it is **caught**
- If the exception is never caught, it crashes the thread it's running on, often crashing the program

Kinds of exceptions

- There are two kinds of exceptions:
  - Checked
  - Unchecked
- Checked exceptions can only be thrown if the throwing code is:
  - Surrounded by a try block with a catch block matching the kind of exception, or
  - Inside a method that is marked with the keyword **`throws`** as throwing the exception in question
- Unchecked exceptions can be thrown at any time (and usually indicate unrecoverable runtime errors)

Examples of exceptions

- Checked exceptions
  - **`FileNotFoundException`**
  - **`IOException`**
  - **`InterruptedException`**
  - Any exception you write that extends **`Exception`**
- Unchecked exceptions
  - **`ArrayIndexOutOfBoundsException`**
  - **`NullPointerException`**
  - **`ArithmeticException`**
  - Any exception you write that extends **`Error`** or **`RuntimeException`**

Exceptions in code

```
// try-with-resources automatically closes Scanner
try (Scanner in = new Scanner(file)) {
  while (in.hasNextInt()) {
     process(in.nextInt());
  }
} catch (FileNotFoundException e) {
     System.out.println("File " +
     file.getName() + " not found !");
}
```

OOP

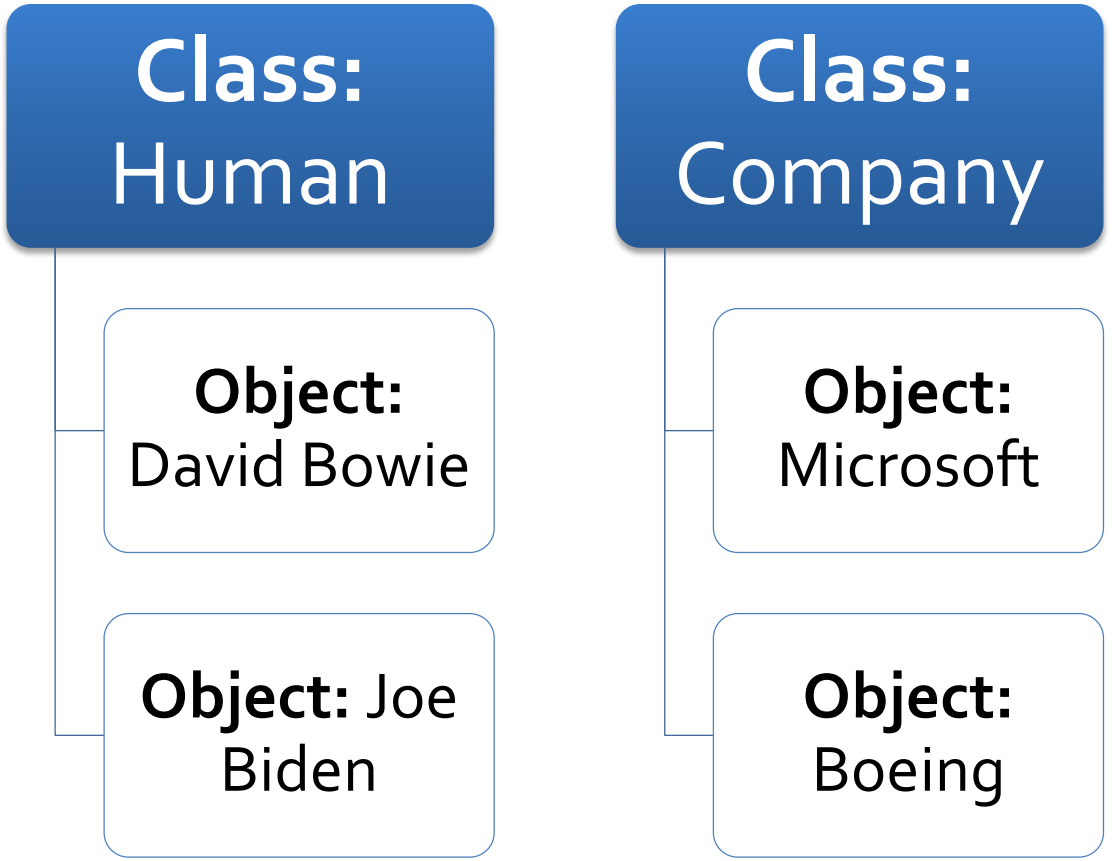
What is an object?

- Members
- Methods
- Why are they useful?

- Code and data together
- Code separated, but data shared
- Code and data separated

What is a class?

- A template or prototype for an object



Object-Oriented Programming

- Encapsulation
- Dynamic dispatch
- Polymorphism
- Inheritance
- Self-reference

Encapsulation

- Information hiding
- We want to bind operations and data tightly together
- Consequently, we don't want you to touch our privates
- Encapsulation in Java is provided by the **`private`** and **`protected`** keywords (and also by default, package level access)
- Hardcore OOP people think that **all** data should be private and most methods should be public

Encapsulation example

```
public class A {
  private int a;

  public int getA() {
     return a;
  }

  public void setA(int value) {
     a = value;
  }
}
```

Inheritance

- Allows code reuse
- Is thought of as an **is-a** relationship
- Java does not allow multiple inheritance, but some languages do
- Deriving a subclass usually means creating a "refined" or "more specific" version of a superclass

Inheritance example

```
public class B extends A {
  // Has member and methods from A
}

public class C extends A {
  // Has A stuff and more
  private int c;
  public int getC(){ return c; }
  public void increment() { ++c; }
}
```

Polymorphism

- A confusing word whose underlying concept many programmers misunderstand
- Polymorphism is when code is designed for a superclass but can be used with a subclass
- If **`BMW235i`** is a subtype of **`Car`**, then you can use an **`BMW235i`** anywhere you could use a **`Car`**

Polymorphism example

```
// Defined somewhere
public void drive( Car car ) {
  …
}

public class BMW235i extends Car {
  …
}

Car car = new Car();
BMW235i bimmer = new BMW235i();
drive(bimmer); // okay
drive(car);  // okay
```

Dynamic dispatch

- Polymorphism can be used to extend the functionality of an existing method using dynamic dispatch
- In dynamic dispatch, the method that is actually called is not known until run time

Dynamic dispatch example

```
public class A {
  public void print()    {
     System.out.println("A");
  }
}

public class B extends A {
  @Override
  public void print() {
     System.out.println("B");
  }
}
```

Dynamic dispatch example

```
A a = new A();
B b = new B(); // B extends A
A c;

a.print(); // A
b.print(); // B

c = a;
c.print(); // A
c = b;
c.print(); // B
```

Self-reference

- Objects are able to refer to themselves
- This can be used to explicitly reference variables in the class
- Or it can be used to provide the object itself as an argument to other methods

Self reference example

```
public class Stuff {
  private int things;

  public void setThings(int things) {
     this.things = things;
  }
}
```

Self reference example

```
public class SelfAdder {
  public void addToList(List list){
     list.add(this);
  }
}
```

Constructor syntax

- Java provides syntax that allows you to call another constructor from the current class or specify which superclass constructor you want to call
- The first line of a constructor is a call to the superclass constructor
- If neither a **`this()`** or a **`super()`** constructor are the first line, an implicit default **`super()`** constructor is called

Constructor example

```
public class A {
  private double half;
  public A(int value){
     half = value / 2.0;
  }
}

public class B extends A {
  public B(int input) {
     super(input); // calls super constructor
  }

  public B() {
     this(5); // calls other constructor
  }
}
```

Interfaces

Interface basics

- An **interface** is a set of methods which a class must have
- **Implementing** an interface means making a promise to define each of the listed methods
- It can do what it wants inside the body of each method, but it must have them to compile
- Unlike superclasses, a class can implement as many interfaces as it wants

Interface definition

- An interface looks a lot like a class, but all its methods are empty
- Interfaces have no members except for (**`static final`**) constants

```
public interface Guitarist {
    void strumChord(Chord chord);
    void playMelody(Melody notes);
}
```

Interface use

```
public class RockGuitarist extends RockMusician
implements Guitarist {

    public void strumChord(Chord chord) {
        System.out.print("Totally wails on that " +
        chord.getName() + " chord!");
    }

    public void playMelody(Melody notes) {
        System.out.print("Burns through the notes " +
        notes.toString() + " like Jimmy Page!" );
    }
}
```

Usefulness

- A class has an **is-a** relationship with interfaces it implements, just like a superclass it extends
- Code that specifies a particular interface can use any class that implements it

```
public static void perform(Guitarist
    guitarist, Chord chord, Melody notes) {
    System.out.println("Give it up " +
        "for the next guitarist!");
    guitarist.strumChord(chord);
    guitarist.playMelody(notes);
}
```

Upcoming

Next time…

- Generics
- Java Collections Framework
- Computational complexity
- Read section 1.4

Reminders

- Read section 1.4
- Work on Assignment 1
  - Due next Friday before midnight!
- Start on Project 1
- **No class Monday!**