BIM492 Design Patterns

ANADOLU UNIVERSITY

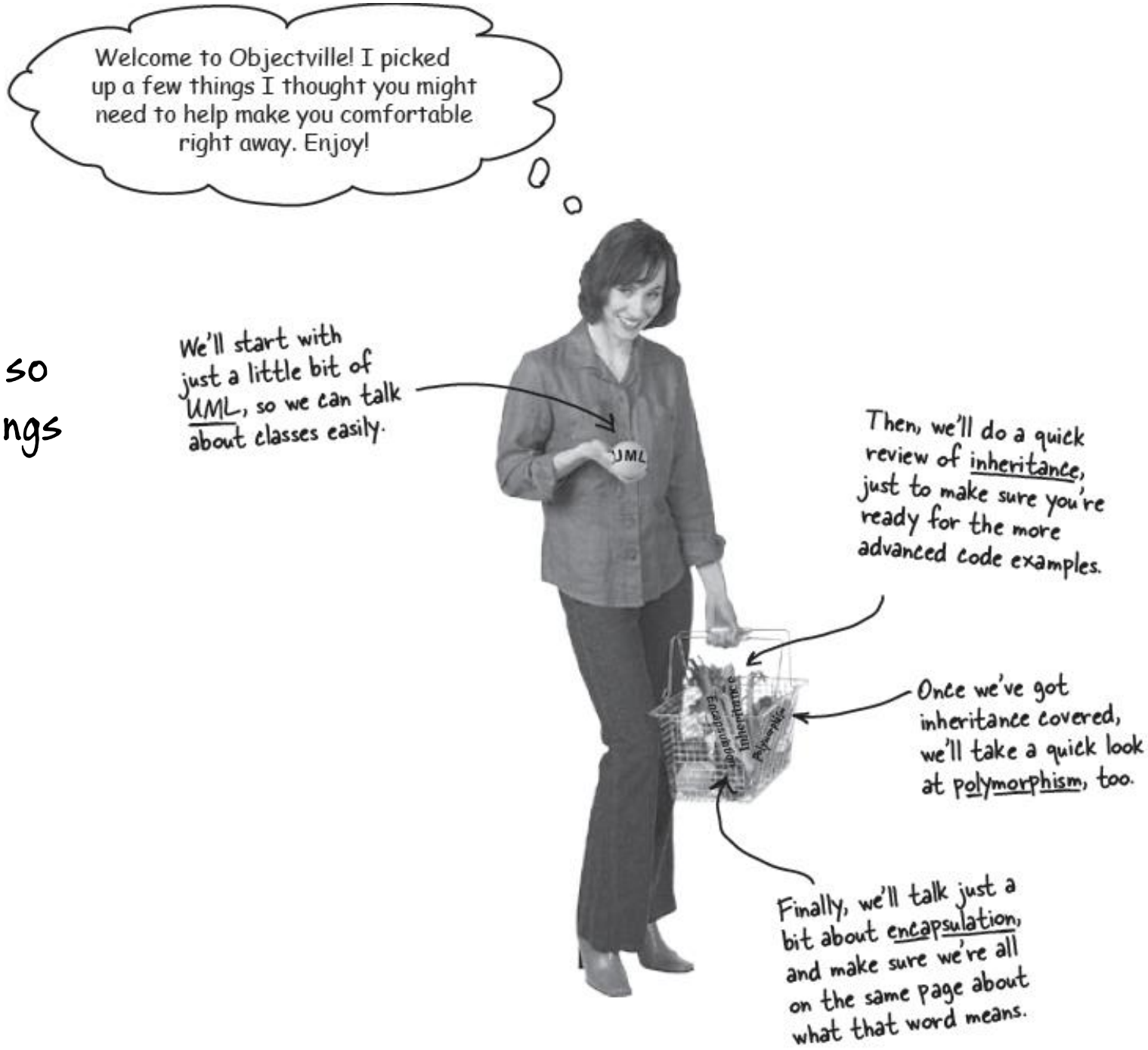
0.5. Welcome to Objectville

Speaking the Language of OO

Welcome to Objectville

Whether this is your first trip to Objectville, or you've visited before, there's no place quite like it.

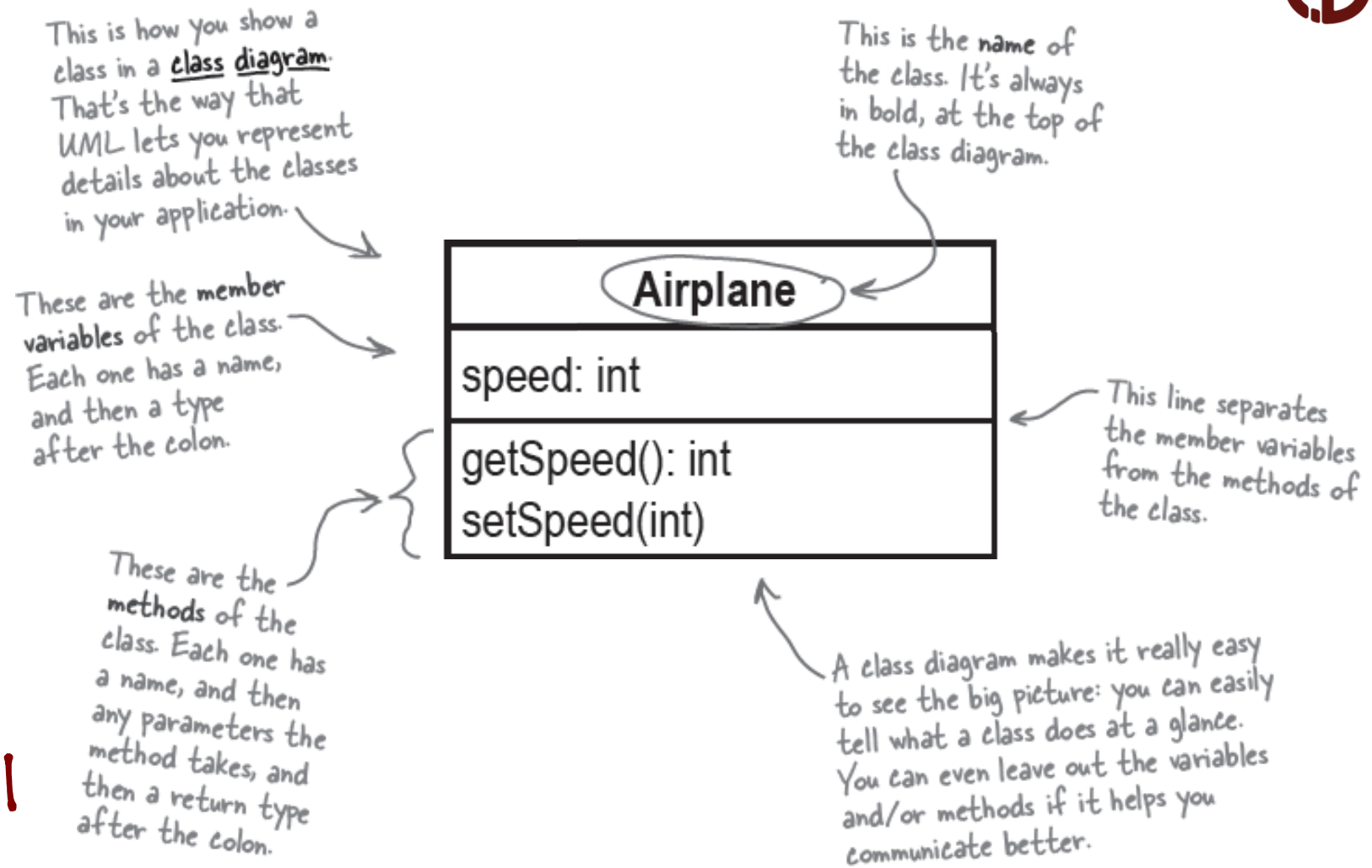
But things are a little different here, so we're here to help you get your bearings before we dive in..



UML and class diagrams

We're going to talk about classes and objects a lot in this class, but it's pretty hard to look at 200 lines of code and focus on the big picture.

So we'll be using UML, the ***Unified Modeling Language***, which is a language used to communicate just the **details** about your **code** and **application's structure** that other developers and customers **need**, without getting details that *aren't* necessary.



Sharpen your pencil

Write the skeleton for the Airplane class.

Using the class diagram above, see if you can write the basic skeleton for the Airplane class. Did you find anything that the class diagram leaves out?

Sharpen your pencil answers

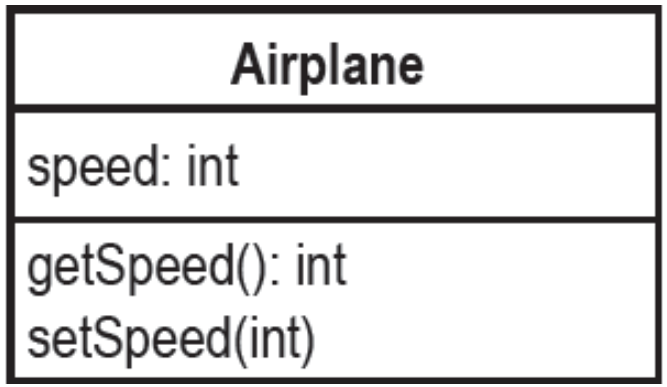
```
public class Airplane {
  private int speed;

  public Airplane() { }

  public void setSpeed(int speed) {
    this.speed = speed;
  }

  public int getSpeed() {
    return speed;
  }
}
```

The class diagram didn't tell us if speed should be public, private, or protected.

Actually, class diagams can provide this information, but in most cases, it's not needed for clear communication.

There was nothing about a constructor in the class diagram. You could have written a constructor that took in an initial speed value, and that would be OK, too.

The class diagram didn't tell us what this method did... we made some assumptions, but we can't be sure if this code is really what was intended.



Next up: inheritance

Jet is called a subclass of Airplane. Airplane is the superclass for Jet.

Jet extends from the Airplane class. That means it inherits all of Airplane's behavior to use for its own.

```
public class Jet extends Airplane {
  private static final int MULTIPLIER = 2;

  public Jet() {
    super();
  }

  public void setSpeed(int speed) {
    super.setSpeed(speed * MULTIPLIER);
  }

  public void accelerate() {
    super.setSpeed(getSpeed() * 2);
  }
}
```

The subclass can add its own variables to the ones that it inherits from Airplane.

super is a special keyword. It refers to the class that this class has inherited behavior from. So here, this calls the constructor of Airplane, Jet's superclass.

The subclass can change the behavior of its superclass, as well as call the superclass's methods. This is called overriding the superclass's behavior.

A subclass can add its own methods to the methods it inherits from its superclass.

Jet also inherits the getSpeed() method from Airplane. But since Jet uses the same version of that method as Airplane, we don't need to write any code to change that method. Even though you can't see it in Jet, it's perfectly OK to call getSpeed() on Jet.

You can call super.getSpeed(), but you can also just call getSpeed(), just as if getSpeed() were a normal method defined in Jet.

Pool Puzzle

Your *job* is to take code snippets from the pool below and place them into the blank lines in the code you see on the right.
You **may** use the same snippet more than once, and you won't need to use all the snippets.
Your *goal* is to create a class that will compile, run, and produce the output.

```
File Edit Window Help LeavingOnAJetplane
%java FlyTest
212
844
1688
6752
13504
27008
1696
```

```
public class FlyTest {
  public static void main(String[] args) {
    Airplane biplane = new Airplane();
    biplane.setSpeed(_____);
    System.out.println(_____________________);
    Jet boeing = new Jet();
    boeing.setSpeed(_____);
    System.out.println(_____________________);
    __________;
    while (______) {
      _____________________;
      System.out.println(_____________________);
      if (_____________________ > 5000) {
        _____________________(_____________________ * 2);
      } else {
        _____________________;
      }
      _______;
    }
    System.out.println(_____________________);
  }
}
```

```
int x = 0
boeing.setSpeed
x = 1
x--
x = 0
x < 4
x < 3
108
424
212
x++
biplane.setSpeed
422
}
}
boeing.getSpeed()
x < 5
biplane.accelerate()
boeing.accelerate()
biplane.getSpeed()
```

Pool Puzzle Solution

Your job was to take code snippets from the pool below, and place them into the blank lines in the code you see on the right. You may use the same snippet more than once, and you won't need to use all the snippets. Your goal was to create a class that will compile, run, and produce the output listed.

```
File Edit Window Help LeavingOnAJetplane
%java FlyTest
212
844
1688
6752
13504
27008
1696
```

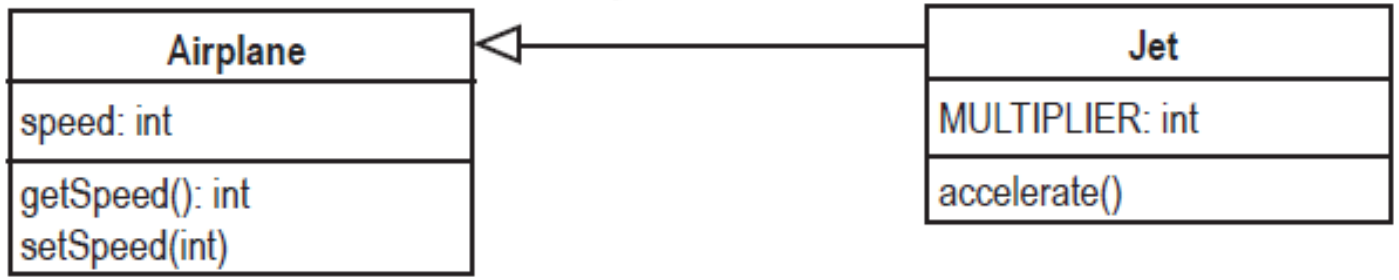
```
public class FlyTest {
  public static void main(String[] args) {
    Airplane biplane = new Airplane();
    biplane.setSpeed( 212 );
    System.out.println(biplane.getSpeed());
    Jet boeing = new Jet();
    boeing.setSpeed( 422 );
    System.out.println(boeing.getSpeed());
    int x = 0 ;
    while ( x < 4 ) {
      boeing.accelerate();
      System.out.println( boeing.getSpeed() );
      if ( boeing.getSpeed()  > 5000) {
        biplane.setSpeed( biplane.getSpeed()* 2);
      } else {
        boeing.accelerate();
      }
      x++ ;
    }
    System.out.println(biplane.getSpeed());
  }
}
```

And polymorphism, too...

- **Polymorphism** is closely related to inheritance.
- When one class inherits from another, then polymorphism allows a **subclass** to **stand in** for the **superclass**.

Here's another class diagram, this time with two classes.

This little arrow means that Jet inherits from Airplane. Don't worry about this notation too much, we'll talk a lot more about inheritance in class diagrams later on.



Jet subclasses Airplane. That means that anywhere that you can use an Airplane...

...you could also use a Jet.

```
Airplane plane = new Airplane();
```

```
Airplane plane = new Jet();
```

So on the left side, you have the superclass...

...and on the right, you can have the superclass OR any of its subclasses.

```
Airplane plane = new Airplane();
```

```
Airplane plane = new Airplane();
Airplane plane = new Jet();
Airplane plane = new Rocket();
```

Last but not least: encapsulation

- **Encapsulation** is **hiding** the implementation of a class so that it is easy to use and easy to change.
- It makes the class act as a black box that provides a service to its users
  - but does not open up the code so someone can change it or use it the wrong way.
- Encapsulation is a key technique in being able to follow the **Open-Closed principle**.

Suppose we rewrote our **Airplane** class like this:

```
public class Airplane {
  public int speed;

  public Airplane() {
  }

  public void setSpeed(int speed) {
    this.speed = speed;
  }

  public int getSpeed() {
    return speed;
  }
}
```

We made the speed variable public, instead of private, and now all parts of your app can access speed directly.

Encapsulation is when you protect information in your code from being used incorrectly.

Now anyone can set the speed directly

This change means that the rest of your app no longer has to call **setSpeed()** to set the speed of a plane; the **speed** variable can be set directly.

```
public class Airplane {
  public int speed;

  public Airplane() {
  }

  public void setSpeed(int speed) {
    this.speed = speed;
  }

  public int getSpeed() {
    return speed;
  }
}
```

```
public class FlyTest2 {
  public static void main(String[] args) {
    Airplane biplane = new Airplane();
    biplane.speed = 212;
    System.out.println(biplane.speed);
  }
}
```

We don't have to use setSpeed() and getSpeed() anymore... we can just access speed directly.

Try this code out... anything surprising in the results you get?

So what's the big deal?

Doesn't seem like much of a problem, does it? But what happens if you create a `Jet` and set its speed like this:

```
public class FlyTest3 {
  public static void main(String[] args) {
    Jet jet1 = new Jet();
    jet1.speed = 212;
    System.out.println(jet1.speed);

    Jet jet2 = new Jet();
    jet2.setSpeed(212);
    System.out.println(jet2.getSpeed());
  }
}
```

Using Jet without encapsulation.

Since Jet inherits from Airplane, you can use the speed variable from its superclass just like it was a part of Jet.

Using Jet with encapsulation

This is how we set and accessed the speed variable when we hid speed from being directly accessed.

Sharpen your pencil

What's the value of encapsulating your data?

Type in, compile, and run the code for `FlyTest3.java`, shown in previous slide. What did your output look like? Write the two lines of output in the blanks below:

Speed of jet1: 212

Speed of jet2: 424

What do you think happened here? Write down why you think you got the speeds that you did for each instance of Jet:

In the Jet class, setSpeed() takes the value supplied, and multiplies it by two before setting the speed of the jet. When we set the speed variable manually, it didn't get multiplied by two.

Finally, summarize what you think the value of encapsulation is:

Encapsulation protects data from being set in an improper way. With encapsulated data, any calculations or checks that the class does on the data are preserved, since the data can't be accessed directly.

So encapsulation does more than just hide information; it makes sure the methods you write to work with your data are actually used!

Let's formally define encapsulation

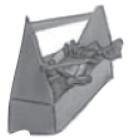
Taking a college class in programming? Here's the official definition of encapsulation... if you're taking an exam, this is the definition to use.

the Scholar's Corner

encapsulation. The process of enclosing programming elements inside larger, more abstract entities. Also known as information hiding, or separation of concerns.

Encapsulation separates your data from your app's behavior.

Then you can control how each part is used by the rest of your application.



Tools for your toolbox

Let's look at the tools you've put in your OOA&D toolbox.

Bullet Points

- UML stands for the Unified Modeling Language.
- UML helps you communicate the structure of your application to other developers, customers, and managers.
- A class diagram gives you an overview of your class, including its methods and variables.
- Inheritance is when one class extends another class to reuse or build upon the inherited class's behavior.
- In inheritance, the class being inherited from is called the superclass; the class that is doing the inheritance is called the subclass.
- A subclass gets all the behavior of its superclass automatically.
- A subclass can override its superclass's behavior to change how a method works.
- Polymorphism is when a subclass "stands in" for its superclass.
- Polymorphism allows your applications to be more flexible, and less resistant to change.
- Encapsulation is when you separate or hide one part of your code from the rest of your code.
- The simplest form of encapsulation is when you make the variables of your classes private, and only expose that data through methods on the class.
- You can also encapsulate groups of data, or even behavior, to control how they are accessed.