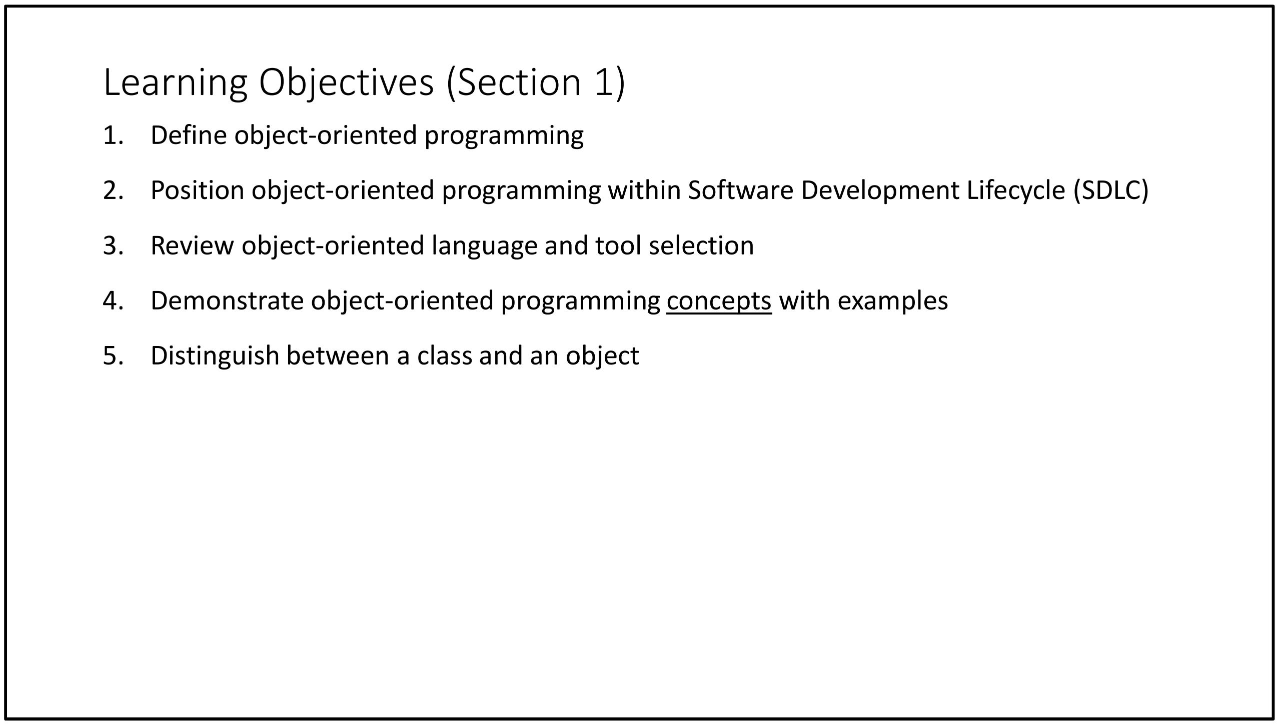
Learning Objectives

1. Define object-oriented programming
2. Position object-oriented programming within Software Development Lifecycle (SDLC)
3. Review object-oriented language and tool selection
4. Demonstrate object-oriented programming concepts with examples
5. Distinguish between a class and an object
6. Identify and define “six” object-oriented concepts
7. Identify the superclass and the subclass in an inheritance relationship
8. Demonstrate inheritance, ownership, and abstraction in snippets of Java code
9. Distinguish between aggregation and composition
10. Depict classes and their relationships using UML class diagrams
11. Define and discuss common object-oriented design patterns
12. Define and discuss common object-oriented design principles and characteristics of bad software
13. Describe how object-oriented programming is fundamentally different
14. Justify the choice to use an object-oriented approach in developing software

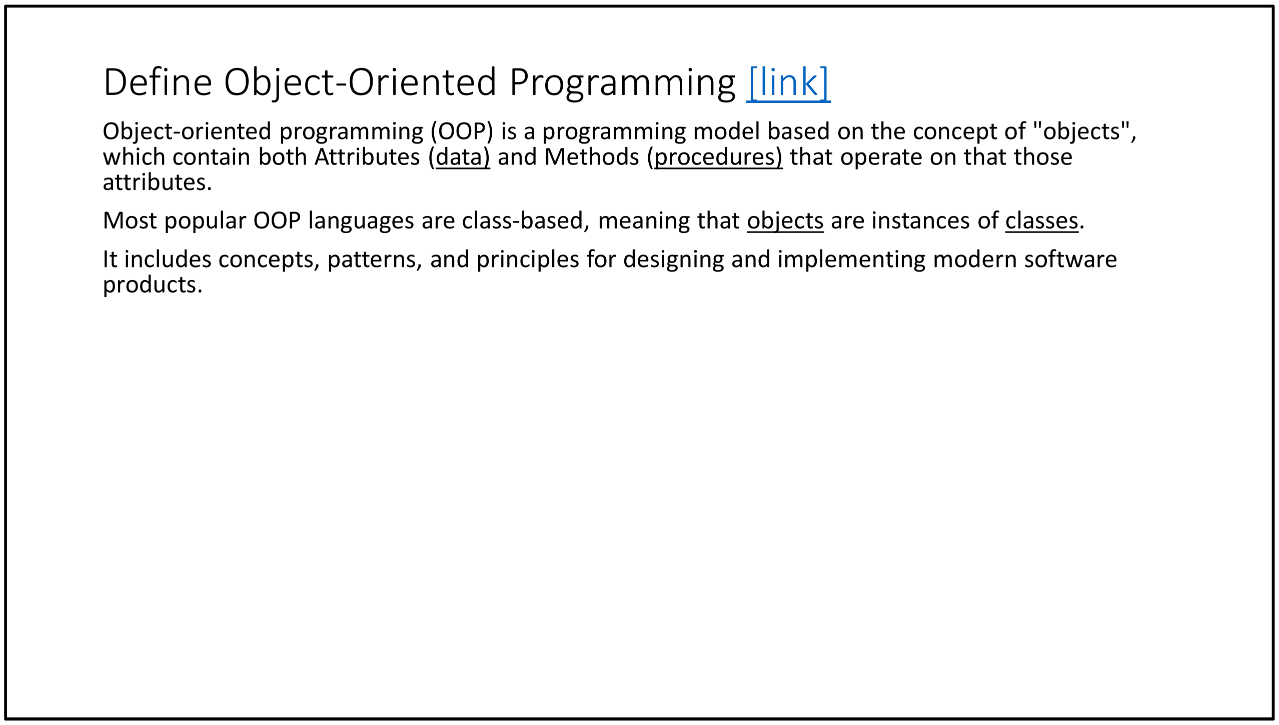
You will notice very quickly that I prefer examples and actual source code over more philosophical discussion. Please feel free to ask for more development philosophy and background if you desire more.



Learning Objectives (Section 1)

1. Define object-oriented programming
2. Position object-oriented programming within Software Development Lifecycle (SDLC)
3. Review object-oriented language and tool selection
4. Demonstrate object-oriented programming <u>concepts</u> with examples
5. Distinguish between a class and an object

You will notice very quickly that I prefer examples and actual source code over more philosophical discussion. Please feel free to ask for more development philosophy and background if you desire more.



Define Object-Oriented Programming [link]

Object-oriented programming (OOP) is a programming model based on the concept of "objects", which contain both Attributes (data) and Methods (procedures) that operate on that those attributes.

Most popular OOP languages are class-based, meaning that objects are instances of classes.

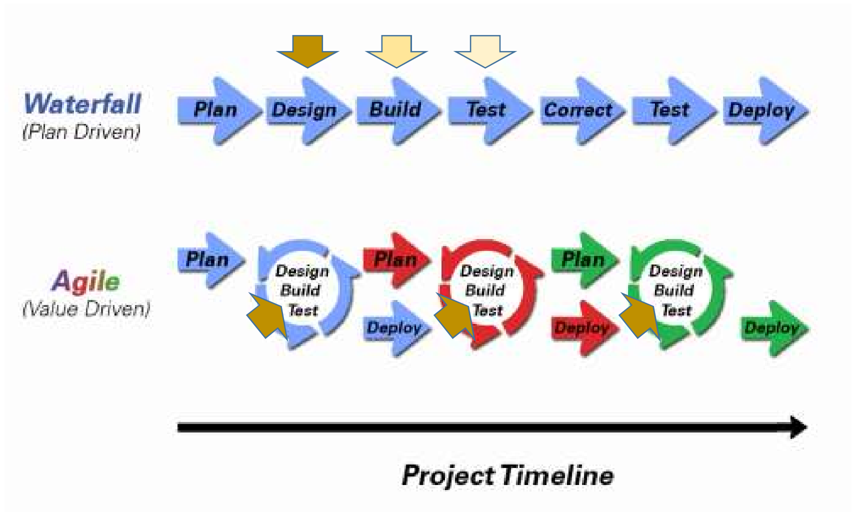
It includes concepts, patterns, and principles for designing and implementing modern software products.

I will often start with definitions from Wikipedia and other sources. If you are struggling with a topic and/or would like more information, it can often be valuable to review the references.

Words and terminology our important as we discuss and learn new concepts. You will notice that I will often struggle with the attribute vs. data and method vs. procedure distinction when I am talking. I would like for us to try to make that distinction in our work as we go through the term.

Concepts – powerful features that prove indispensable to modern software development, brought to us automatically by object-oriented programming.
Patterns – tried-and-true templates for forging relationships between classes
Principles – guidelines that help you determine what classes are needed and how they should divide up the work

Position Object-Oriented Programming within Various Development Methodologies



Development Methodology and Software Development Lifecycle (SDLC) are often used interchangeably.
The Iterative development methodology is not depicted here as even the mainstays and inventors of the Iterative development methodology seem to be moving toward agile. Plus as Waterfall “holdouts” move, they seem to be moving directly toward Agile.

Development Methodologies (SDLCs) are a future Bonus Topic. There is a optional slide and notes at the end of this deck.

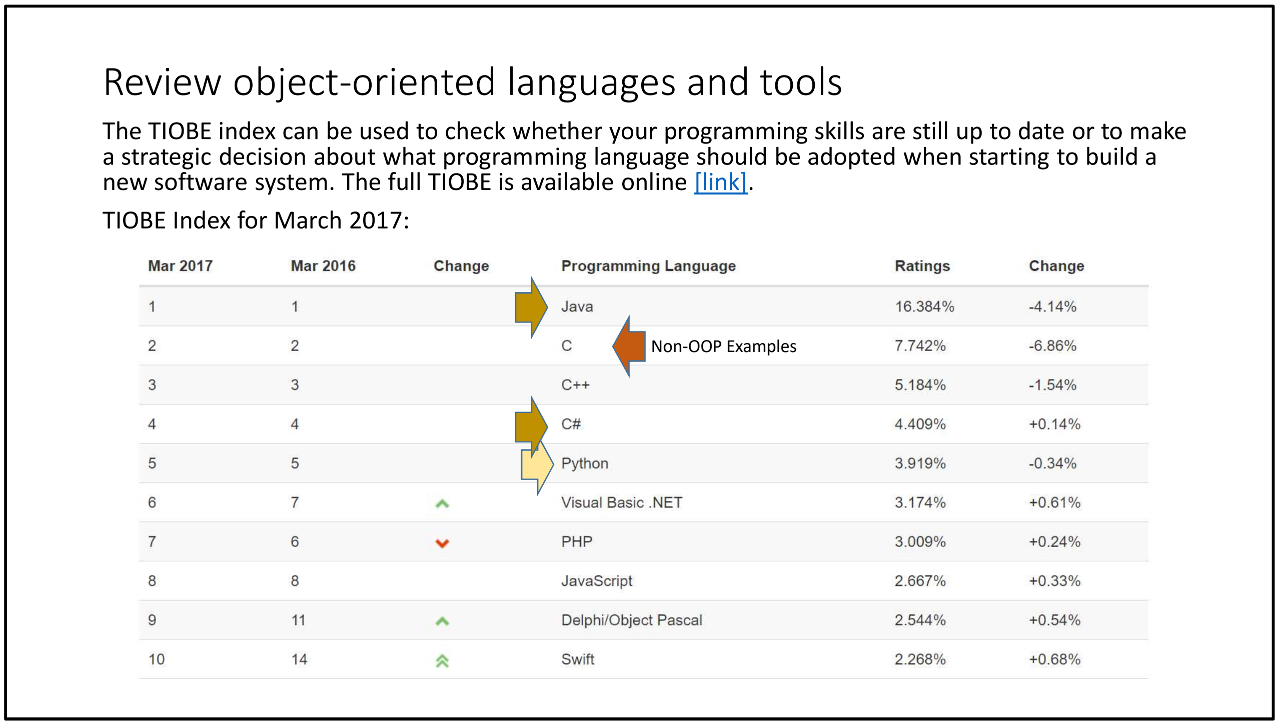
Object oriented programming practices evolve and reprioritize depending on the development

In Waterfall (as well as in Iterative) object-oriented design often play a critical role in the (big upfront) design activities. UML diagrams and project artifacts are often important to the overall project success.

Practical reality has been that these design artifacts often do not reflect the actual implementation and are rarely maintained or updated.

The Agile practitioners do not reject these design artifacts. However, the focus on shorter time horizons, evolving architecture, and working code changes the value proposition for object-oriented practices to more focus on the build, test, enhance activities.

More later on Development Methodologies



Review object-oriented languages and tools

The TIOBE index can be used to check whether your programming skills are still up to date or to make a strategic decision about what programming language should be adopted when starting to build a new software system. The full TIOBE is available online [link].

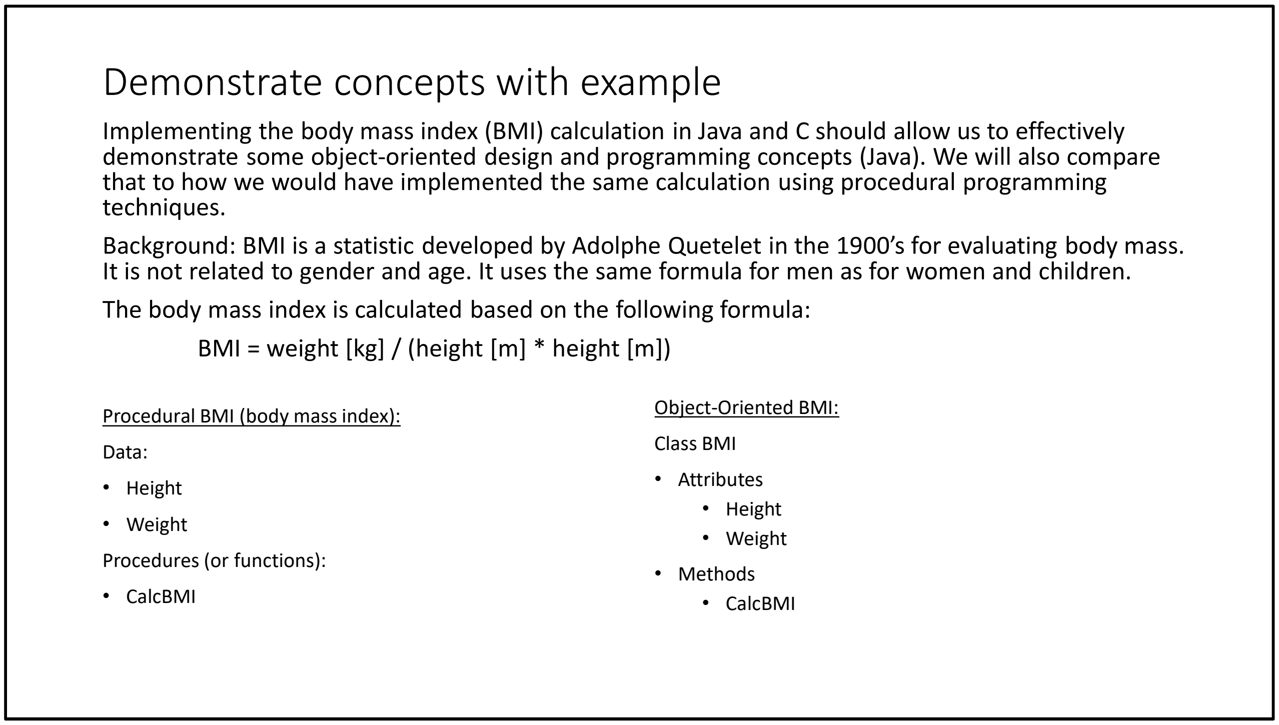
TIOBE Index for March 2017:

Mar 2017	Mar 2016	Change	Programming Language	Ratings	Change
1	1		Java	16.384%	-4.14%
2	2		C	7.742%	-6.86%
3	3		C++	5.184%	-1.54%
4	4		C#	4.409%	+0.14%
5	5		Python	3.919%	-0.34%
6	7	▲	Visual Basic .NET	3.174%	+0.61%
7	6	▼	PHP	3.009%	+0.24%
8	8		JavaScript	2.667%	+0.33%
9	11	▲	Delphi/Object Pascal	2.544%	+0.54%
10	14	▲▲	Swift	2.268%	+0.68%

Non-OOP Examples

We will utilize mostly Java and C# for our object-oriented programming examples. We may or may not do any Python work. Since it is often 'unnatural' to show procedural programming examples in Java, C#, or Python, we will implement programs in C to demonstrate procedure programming examples.

Note that our reluctance to utilize C++ as a OOP learning tool is does not diminish the value of the C++ toolset. However, C++ is generally considered a very powerful set of tools with a steep learning curve. It's a sharp knife… use it carefully.



Demonstrate concepts with example

Implementing the body mass index (BMI) calculation in Java and C should allow us to effectively demonstrate some object-oriented design and programming concepts (Java). We will also compare that to how we would have implemented the same calculation using procedural programming techniques.

Background: BMI is a statistic developed by Adolphe Quetelet in the 1900's for evaluating body mass. It is not related to gender and age. It uses the same formula for men as for women and children.

The body mass index is calculated based on the following formula:

BMI = weight [kg] / (height [m] * height [m])

Procedural BMI (body mass index):

Data:

- Height
- Weight

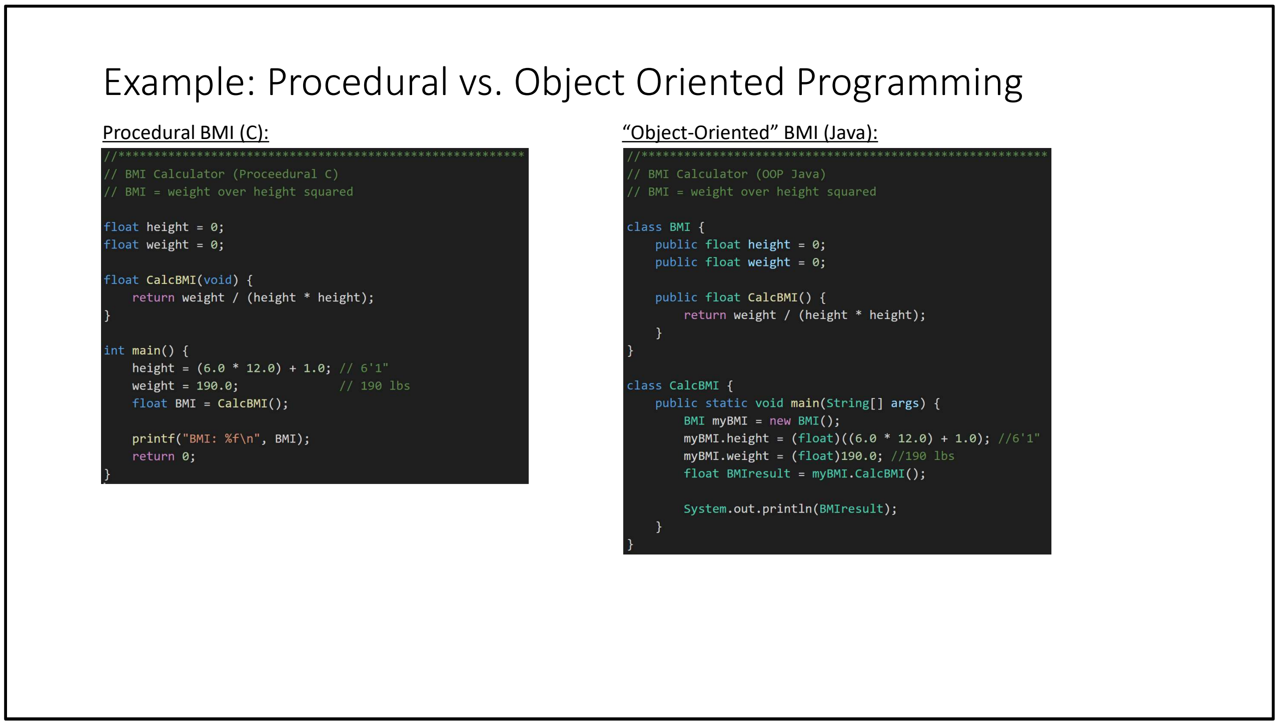
Procedures (or functions):

- CalcBMI

Object-Oriented BMI:

Class BMI

- Attributes
    - Height
    - Weight
- Methods
    - CalcBMI



Example: Procedural vs. Object Oriented Programming

Procedural BMI (C):

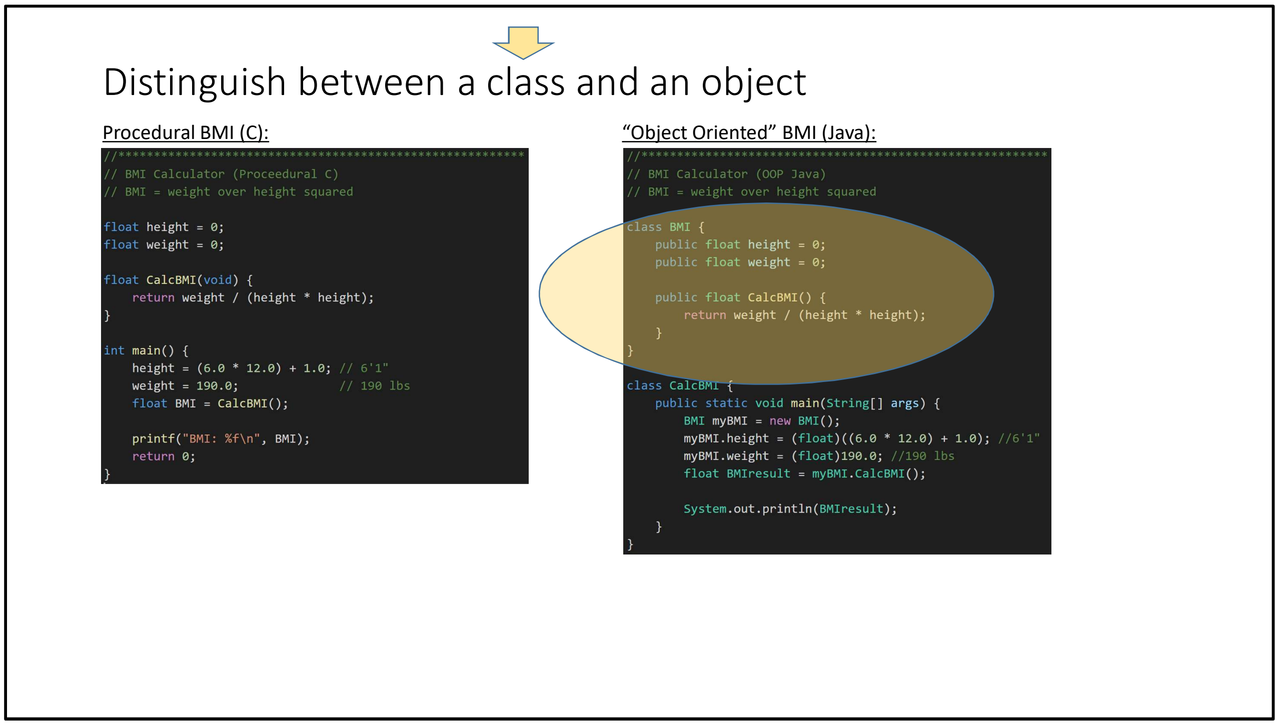
```c
//*********************************************************
// BMI Calculator (Proceedural C)
// BMI = weight over height squared

float height = 0;
float weight = 0;

float CalcBMI(void) {
    return weight / (height * height);
}

int main() {
    height = (6.0 * 12.0) + 1.0; // 6'1"
    weight = 190.0;              // 190 lbs
    float BMI = CalcBMI();

    printf("BMI: %f\n", BMI);
    return 0;
}
```

"Object-Oriented" BMI (Java):

```java
//*********************************************************
// BMI Calculator (OOP Java)
// BMI = weight over height squared

class BMI {
    public float height = 0;
    public float weight = 0;

    public float CalcBMI() {
        return weight / (height * height);
    }
}

class CalcBMI {
    public static void main(String[] args) {
        BMI myBMI = new BMI();
        myBMI.height = (float)((6.0 * 12.0) + 1.0); //6'1"
        myBMI.weight = (float)190.0; //190 lbs
        float BMIresult = myBMI.CalcBMI();

        System.out.println(BMIresult);
    }
}
```

We have quite a lot of code to look at here. Let's start with the procedural C code. We have data (height & weight) and procedures (CalcBMI),

Do you see the logic issue? Hang onto that for a moment. Fixing that will be our coding exercise to demonstrate some object-oriented concepts.



Distinguish between a class and an object

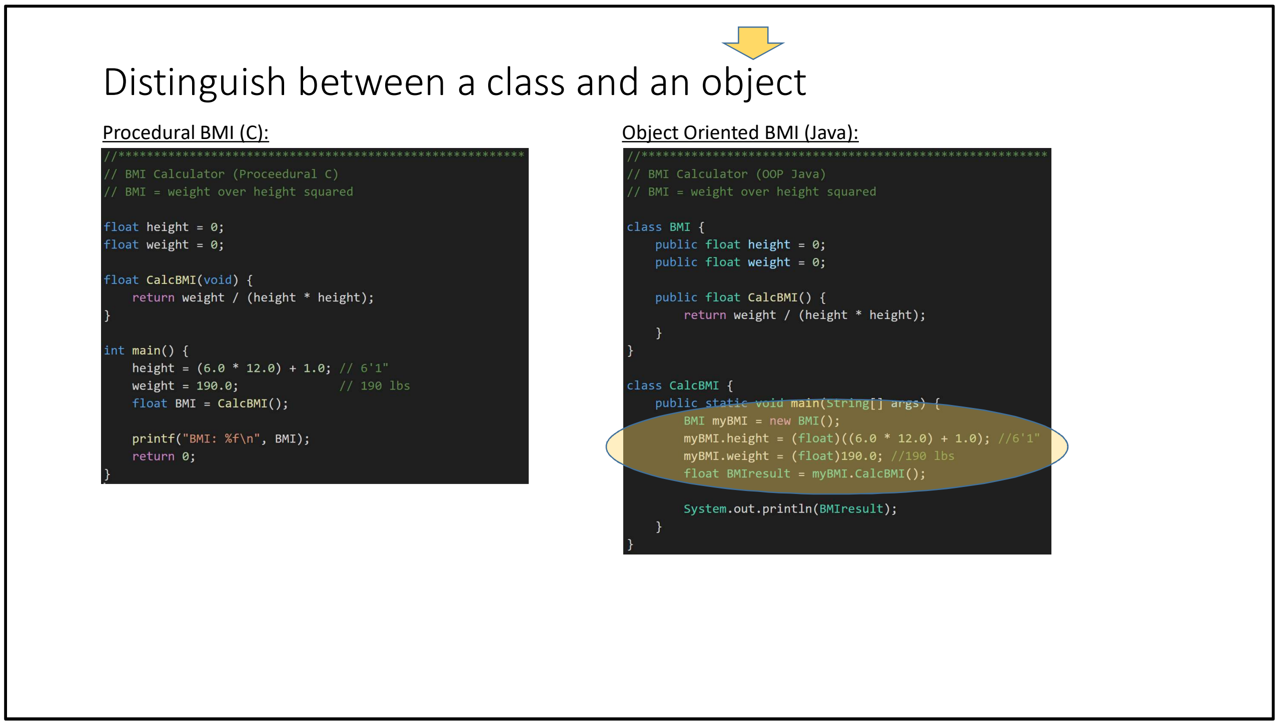
Procedural BMI (C):

```
//*********************************************************
// BMI Calculator (Proceedural C)
// BMI = weight over height squared

float height = 0;
float weight = 0;

float CalcBMI(void) {
    return weight / (height * height);
}

int main() {
    height = (6.0 * 12.0) + 1.0; // 6'1"
    weight = 190.0;              // 190 lbs
    float BMI = CalcBMI();

    printf("BMI: %f\n", BMI);
    return 0;
}
```

"Object Oriented" BMI (Java):

```
//*********************************************************
// BMI Calculator (OOP Java)
// BMI = weight over height squared

class BMI {
    public float height = 0;
    public float weight = 0;

    public float CalcBMI() {
        return weight / (height * height);
    }
}

class CalcBMI {
    public static void main(String[] args) {
        BMI myBMI = new BMI();
        myBMI.height = (float)((6.0 * 12.0) + 1.0); //6'1"
        myBMI.weight = (float)190.0; //190 lbs
        float BMIresult = myBMI.CalcBMI();

        System.out.println(BMIresult);
    }
}
```

There we have it, a Class. It has attributes (height & weight) and a method (CalcBMI). Still has the same logic problem… but let's wait on that.



Distinguish between a class and an object

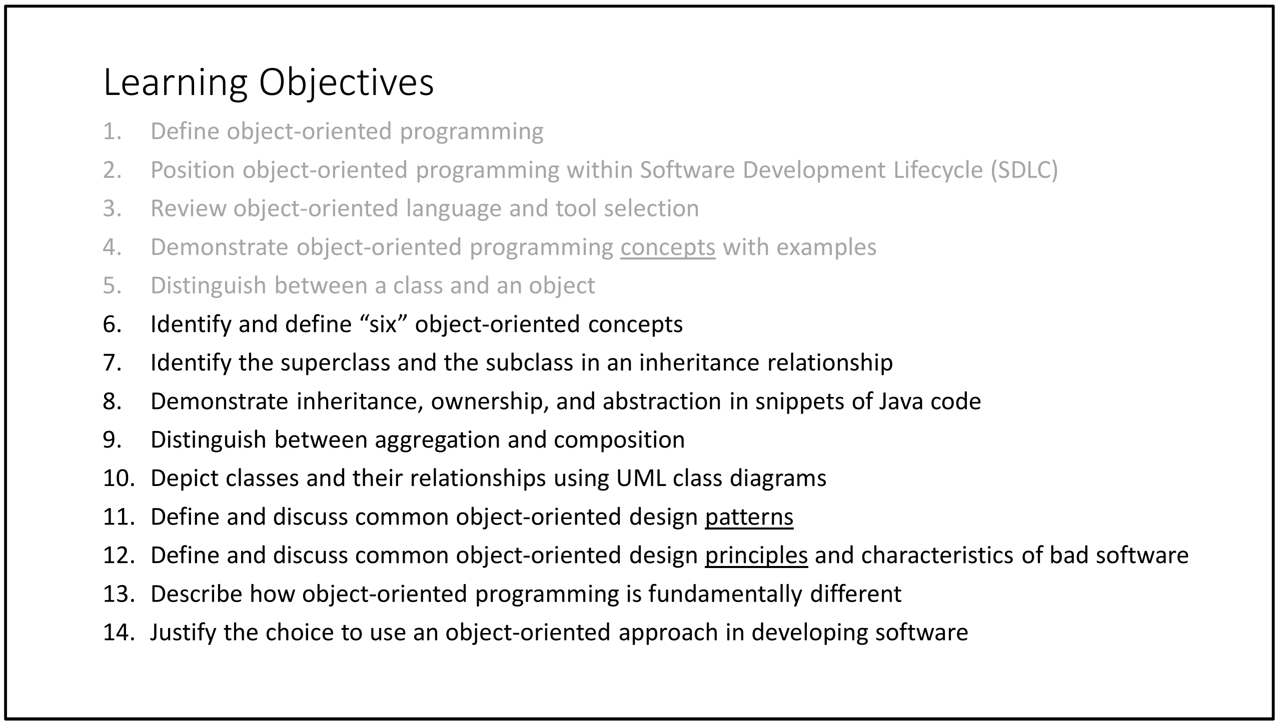
Procedural BMI (C):

```
//*********************************************************
// BMI Calculator (Proceedural C)
// BMI = weight over height squared

float height = 0;
float weight = 0;

float CalcBMI(void) {
    return weight / (height * height);
}

int main() {
    height = (6.0 * 12.0) + 1.0; // 6'1"
    weight = 190.0;              // 190 lbs
    float BMI = CalcBMI();

    printf("BMI: %f\n", BMI);
    return 0;
}
```

Object Oriented BMI (Java):

```
//*********************************************************
// BMI Calculator (OOP Java)
// BMI = weight over height squared

class BMI {
    public float height = 0;
    public float weight = 0;

    public float CalcBMI() {
        return weight / (height * height);
    }
}

class CalcBMI {
    public static void main(String[] args) {
        BMI myBMI = new BMI();
        myBMI.height = (float)((6.0 * 12.0) + 1.0); //6'1"
        myBMI.weight = (float)190.0; //190 lbs
        float BMIresult = myBMI.CalcBMI();

        System.out.println(BMIresult);
    }
}
```

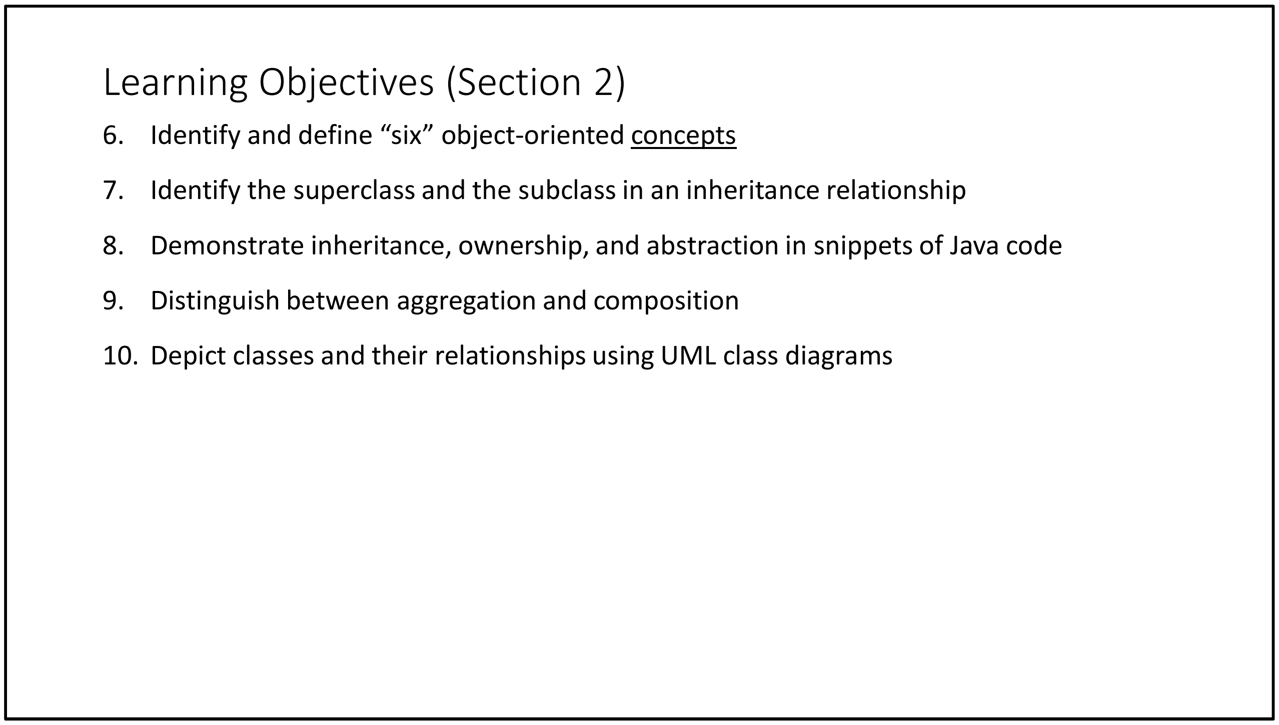
Ahhhh… and we have an object “myBMI’ which is an instance off the class “BMI”.



Learning Objectives

1. Define object-oriented programming
2. Position object-oriented programming within Software Development Lifecycle (SDLC)
3. Review object-oriented language and tool selection
4. Demonstrate object-oriented programming concepts with examples
5. Distinguish between a class and an object
6. Identify and define "six" object-oriented concepts
7. Identify the superclass and the subclass in an inheritance relationship
8. Demonstrate inheritance, ownership, and abstraction in snippets of Java code
9. Distinguish between aggregation and composition
10. Depict classes and their relationships using UML class diagrams
11. Define and discuss common object-oriented design patterns
12. Define and discuss common object-oriented design principles and characteristics of bad software
13. Describe how object-oriented programming is fundamentally different
14. Justify the choice to use an object-oriented approach in developing software

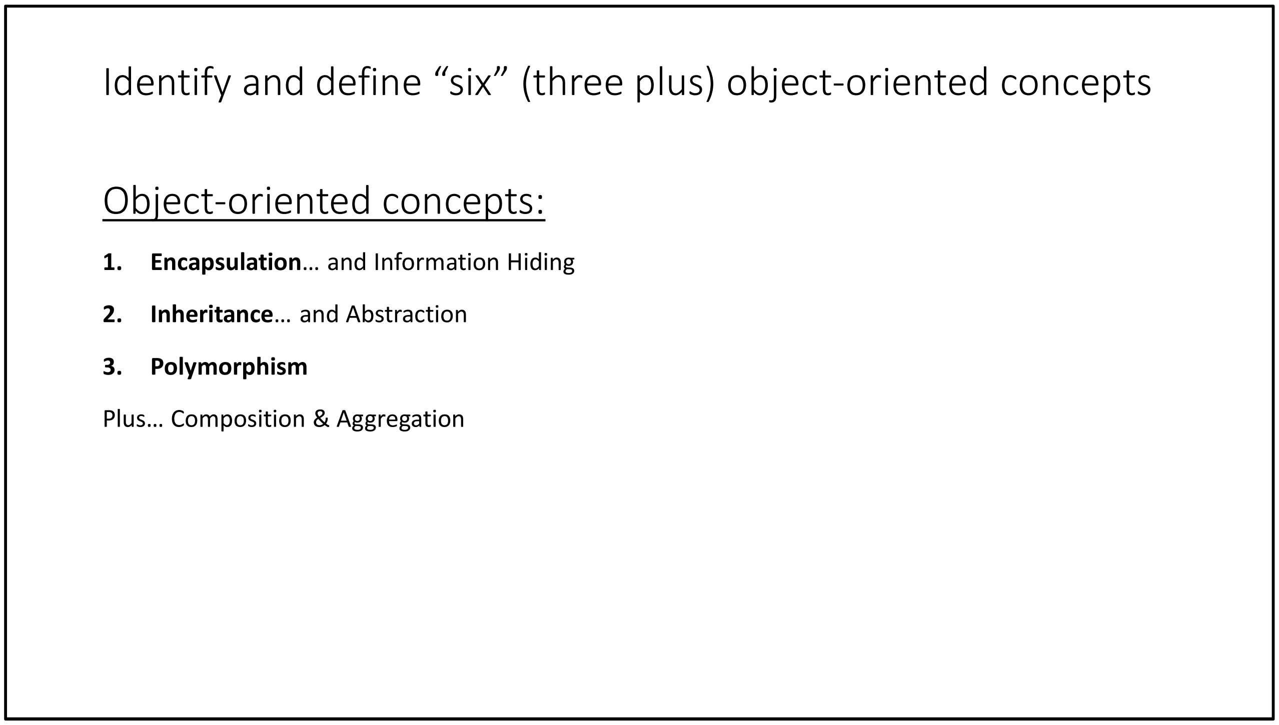
You will notice very quickly that I prefer examples and actual source code over more philosophical discussion. Please feel free to ask for more development philosophy and background if you desire more.



Learning Objectives (Section 2)

6. Identify and define “six” object-oriented concepts
7. Identify the superclass and the subclass in an inheritance relationship
8. Demonstrate inheritance, ownership, and abstraction in snippets of Java code
9. Distinguish between aggregation and composition
10. Depict classes and their relationships using UML class diagrams

You will notice very quickly that I prefer examples and actual source code over more philosophical discussion. Please feel free to ask for more development philosophy and background if you desire more.



Identify and define “six” (three plus) object-oriented concepts

Object-oriented concepts:

1. **Encapsulation**… and Information Hiding
2. **Inheritance**… and Abstraction
3. **Polymorphism**

Plus… Composition & Aggregation

Whenever you are asked a conceptual question about object-programming in an software development interview (and you will be), answer confidently “Encapsulation”, “Inheritance”, and “Polymorphism”.

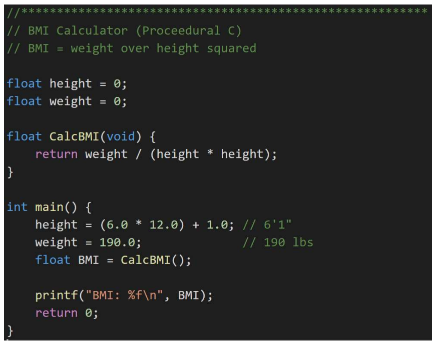
When asked what is Encapsulation (or how would you implement it), say, “I would limit or minimize variable scope and keep data attributes private as often as possible.”

Now as we are going through our object-oriented examples, be thinking about how you would answer the “What is Inheritance?” and “What is Polymorphism?” interview questions. Note that answering them both with very brief and succinct animal examples can be very effective.

Now it would exceptional if we were also able to effectively utilize these concepts after we get the job :-) Let’s start by walking through an example.

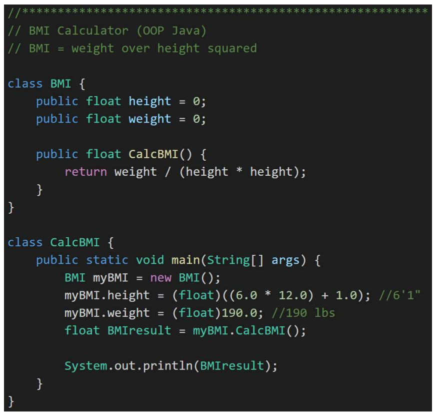
The Problem? *BMI formula assumes M (height) and KG (weight)*

Procedural BMI (C):



```c
//*********************************************************
// BMI Calculator (Proceedural C)
// BMI = weight over height squared

float height = 0;
float weight = 0;

float CalcBMI(void) {
    return weight / (height * height);
}

int main() {
    height = (6.0 * 12.0) + 1.0; // 6'1"
    weight = 190.0;              // 190 lbs
    float BMI = CalcBMI();

    printf("BMI: %f\n", BMI);
    return 0;
}
```

Object Oriented BMI (Java):



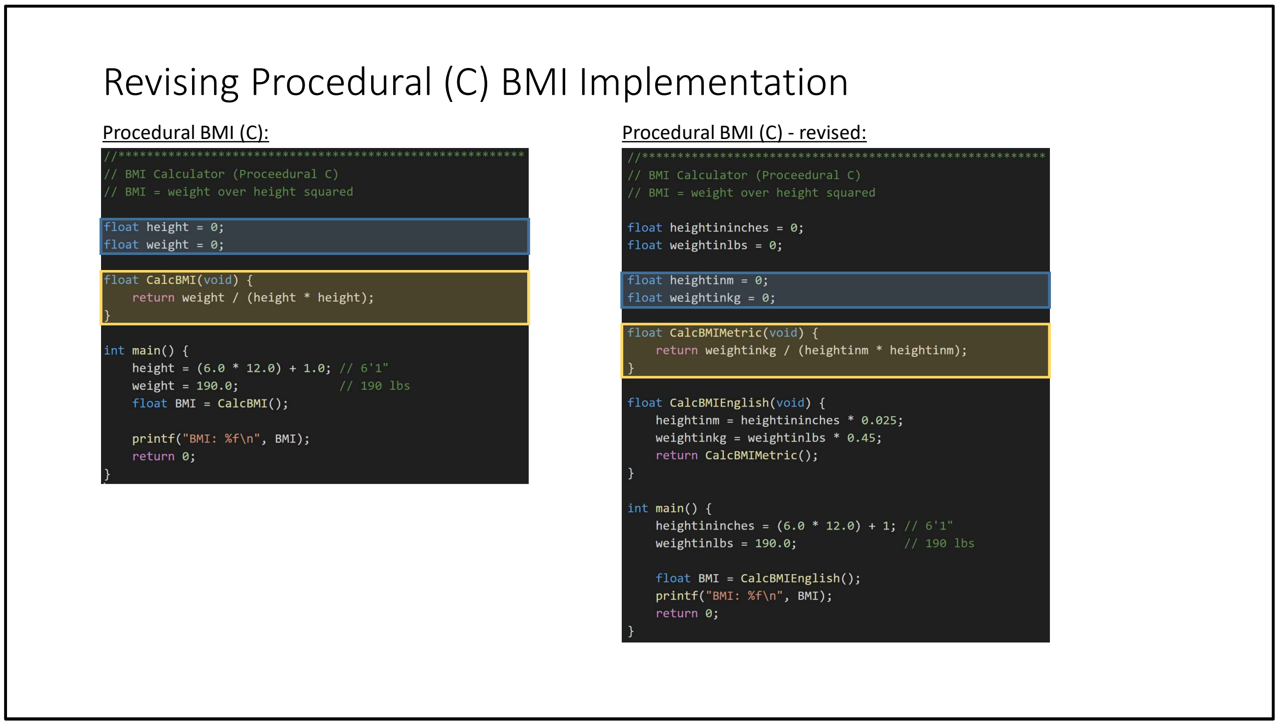
```java
//*********************************************************
// BMI Calculator (OOP Java)
// BMI = weight over height squared

class BMI {
    public float height = 0;
    public float weight = 0;

    public float CalcBMI() {
        return weight / (height * height);
    }
}

class CalcBMI {
    public static void main(String[] args) {
        BMI myBMI = new BMI();
        myBMI.height = (float)((6.0 * 12.0) + 1.0); //6'1"
        myBMI.weight = (float)190.0; //190 lbs
        float BMIresult = myBMI.CalcBMI();

        System.out.println(BMIresult);
    }
}
```

Let's go back to where we left off in our example.

We have a problem in both our procedural (C) and object-oriented (Java) implementations. Our BMI formula assumes metric inputs only (kg & m) our interactions with the BMI procedures and methods are using English units (inches & lbs). So our implementations do work; however, they only work for metric. Let fix our BMI (C) implementation in our normal procedural way.

After that we will enhance our Java implementation using Encapsulation and Inheritance concepts.

Polymorphism will need a different example… one with animals.



Revising Procedural (C) BMI Implementation

Procedural BMI (C):

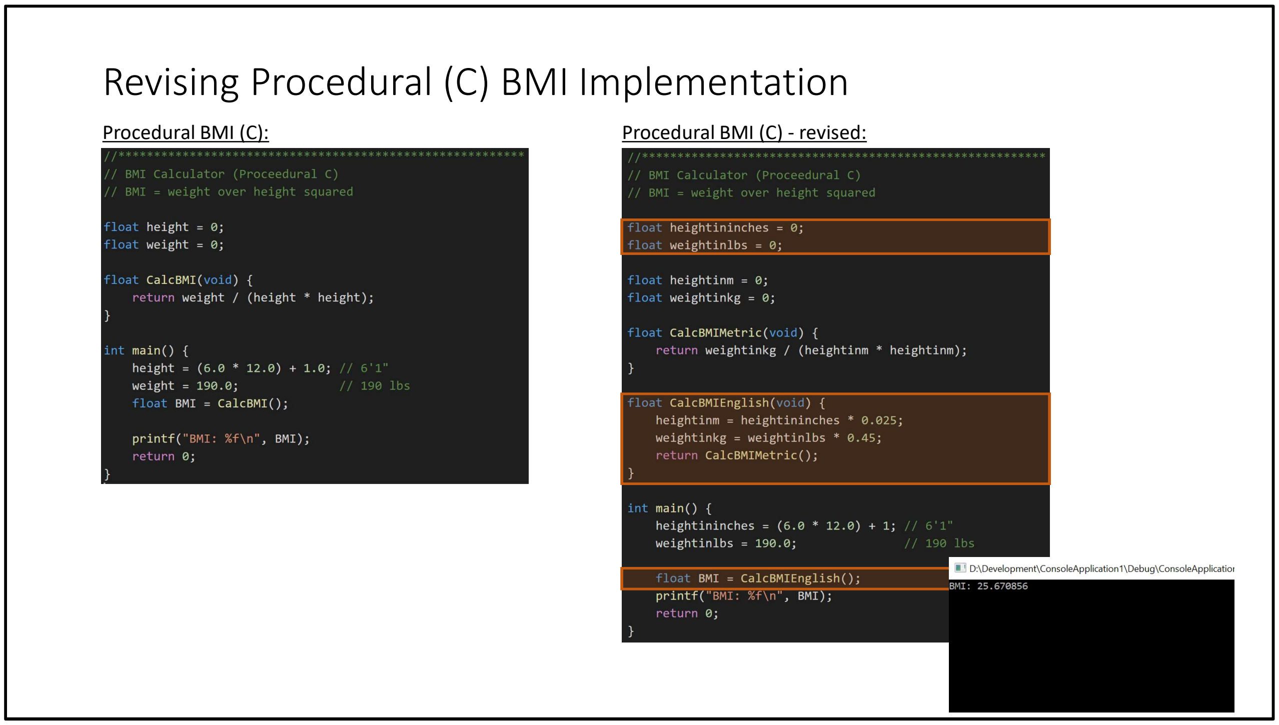
```c
//**********************************************************
// BMI Calculator (Proceedural C)
// BMI = weight over height squared

float height = 0;
float weight = 0;

float CalcBMI(void) {
    return weight / (height * height);
}

int main() {
    height = (6.0 * 12.0) + 1.0; // 6'1"
    weight = 190.0;              // 190 lbs
    float BMI = CalcBMI();

    printf("BMI: %f\n", BMI);
    return 0;
}
```

Procedural BMI (C) - revised:

```c
//**********************************************************
// BMI Calculator (Proceedural C)
// BMI = weight over height squared

float heightininches = 0;
float weightinlbs = 0;

float heightinm = 0;
float weightinkg = 0;

float CalcBMIMetric(void) {
    return weightinkg / (heightinm * heightinm);
}

float CalcBMIEnglish(void) {
    heightinm = heightininches * 0.025;
    weightinkg = weightinlbs * 0.45;
    return CalcBMIMetric();
}

int main() {
    heightininches = (6.0 * 12.0) + 1; // 6'1"
    weightinlbs = 190.0;               // 190 lbs

    float BMI = CalcBMIEnglish();
    printf("BMI: %f\n", BMI);
    return 0;
}
```

Let's go back to where we left off in our example.

We have a problem in both our procedural (C) and object-oriented (Java) implementations. Our BMI formula assumes metric inputs only (kg & m) our interactions with the BMI procedures and methods are using English units (inches & lbs). So our implementations do work; however, they only work for metric. Let fix our BMI (C) implementation (without losing what already works) and then let's enhance our Java implementation using Encapsulation and Inheritance concepts. Polymorphism will need a different example… one with animals.



Revising Procedural (C) BMI Implementation

Procedural BMI (C):

```c
//*********************************************************
// BMI Calculator (Proceedural C)
// BMI = weight over height squared

float height = 0;
float weight = 0;

float CalcBMI(void) {
    return weight / (height * height);
}

int main() {
    height = (6.0 * 12.0) + 1.0; // 6'1"
    weight = 190.0;              // 190 lbs
    float BMI = CalcBMI();

    printf("BMI: %f\n", BMI);
    return 0;
}
```

Procedural BMI (C) - revised:

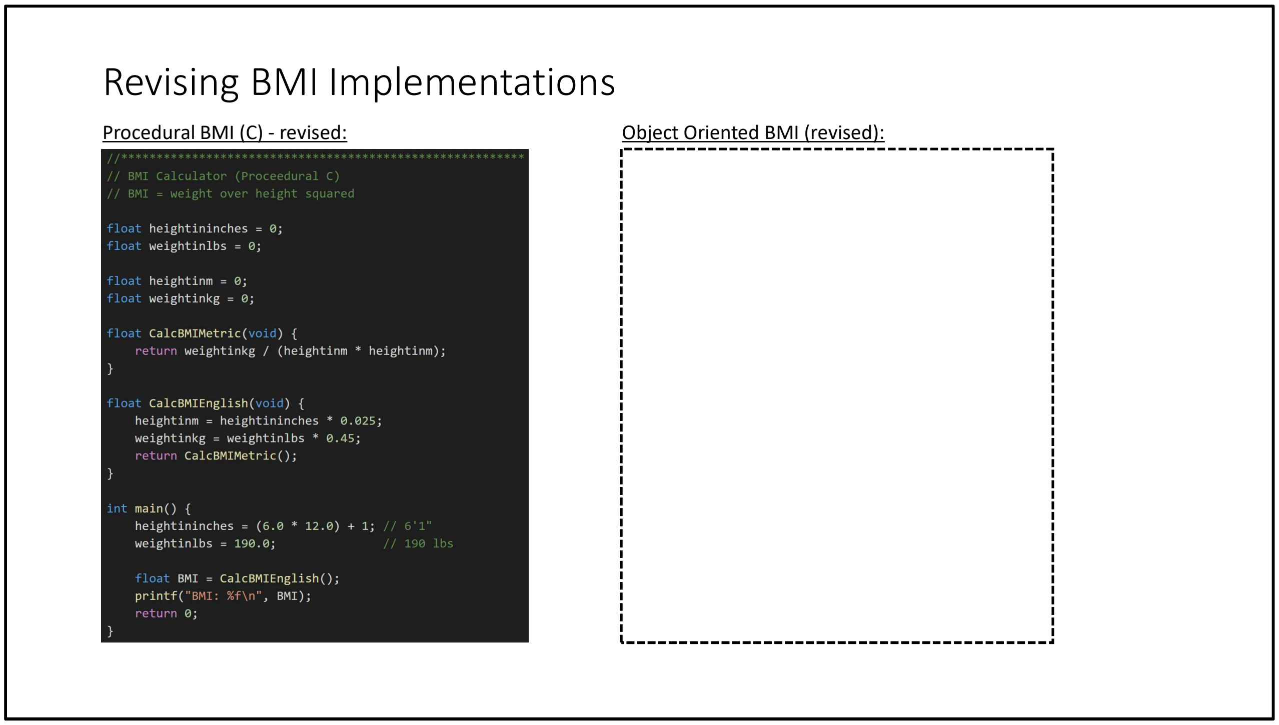
```c
//*********************************************************
// BMI Calculator (Proceedural C)
// BMI = weight over height squared

float heightininches = 0;
float weightinlbs = 0;

float heightinm = 0;
float weightinkg = 0;

float CalcBMIMetric(void) {
    return weightinkg / (heightinm * heightinm);
}

float CalcBMIEnglish(void) {
    heightinm = heightininches * 0.025;
    weightinkg = weightinlbs * 0.45;
    return CalcBMIMetric();
}

int main() {
    heightininches = (6.0 * 12.0) + 1; // 6'1"
    weightinlbs = 190.0;               // 190 lbs

    float BMI = CalcBMIEnglish();
    printf("BMI: %f\n", BMI);
    return 0;
}
```

D:\Development\ConsoleApplication1\Debug\ConsoleApplicatio

```
BMI: 25.670856
```

Let's go back to where we left off in our example.

We have a problem in both our procedural (C) and object-oriented (Java) implementations. Our BMI formula assumes metric inputs only (kg & m) our interactions with the BMI procedures and methods are using English units (inches & lbs). So our implementations do work; however, they only work for metric. Let fix our BMI (C) implementation (without losing what already works) and then let's enhance our Java implementation using Encapsulation and Inheritance concepts. Polymorphism will need a different example… one with animals.



Revising BMI Implementations

<u>Procedural BMI (C) - revised:</u>

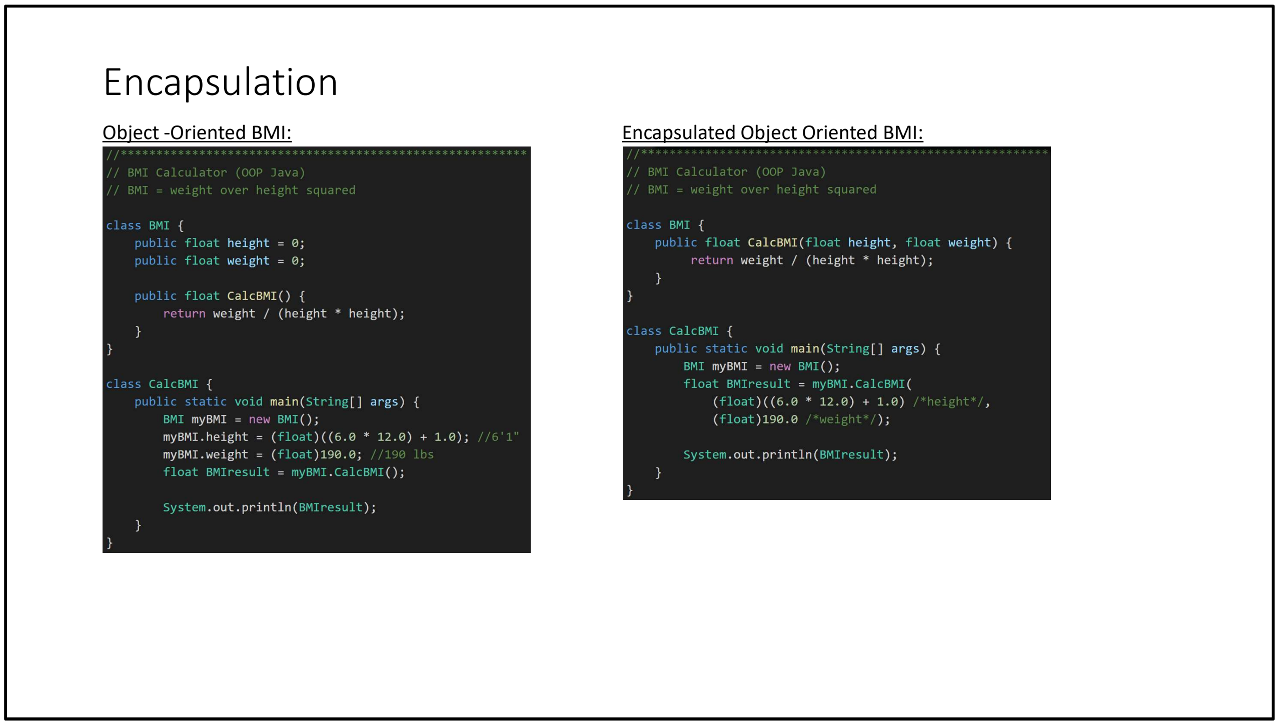
```c
//*********************************************************
// BMI Calculator (Proceedural C)
// BMI = weight over height squared

float heightininches = 0;
float weightinlbs = 0;

float heightinm = 0;
float weightinkg = 0;

float CalcBMIMetric(void) {
    return weightinkg / (heightinm * heightinm);
}

float CalcBMIEnglish(void) {
    heightinm = heightininches * 0.025;
    weightinkg = weightinlbs * 0.45;
    return CalcBMIMetric();
}

int main() {
    heightininches = (6.0 * 12.0) + 1; // 6'1"
    weightinlbs = 190.0;               // 190 lbs

    float BMI = CalcBMIEnglish();
    printf("BMI: %f\n", BMI);
    return 0;
}
```

<u>Object Oriented BMI (revised):</u>

Now consider what it would be like to reuse this C procedural code. Important note: Object Oriented program is NOT necessarily optimized for writing new code. It IS optimized for reusing, supporting, testing, and enhancing code! How would this change if instead of ~20 lines of code, we have a thousand lines of code… or 10,000… or 1,000,000 lines? Software development and testing complexity grows exponentially as the size of the code grows.

Consider: : rigidity, immobility, fragility within the ongoing sdlc (software/systems development lifecycle)

Waterfall, Iterative, and Agile



Encapsulation

Object -Oriented BMI:

```java
//************************************************************
// BMI Calculator (OOP Java)
// BMI = weight over height squared

class BMI {
    public float height = 0;
    public float weight = 0;

    public float CalcBMI() {
        return weight / (height * height);
    }
}

class CalcBMI {
    public static void main(String[] args) {
        BMI myBMI = new BMI();
        myBMI.height = (float)((6.0 * 12.0) + 1.0); //6'1"
        myBMI.weight = (float)190.0; //190 lbs
        float BMIresult = myBMI.CalcBMI();

        System.out.println(BMIresult);
    }
}
```

Encapsulated Object Oriented BMI:

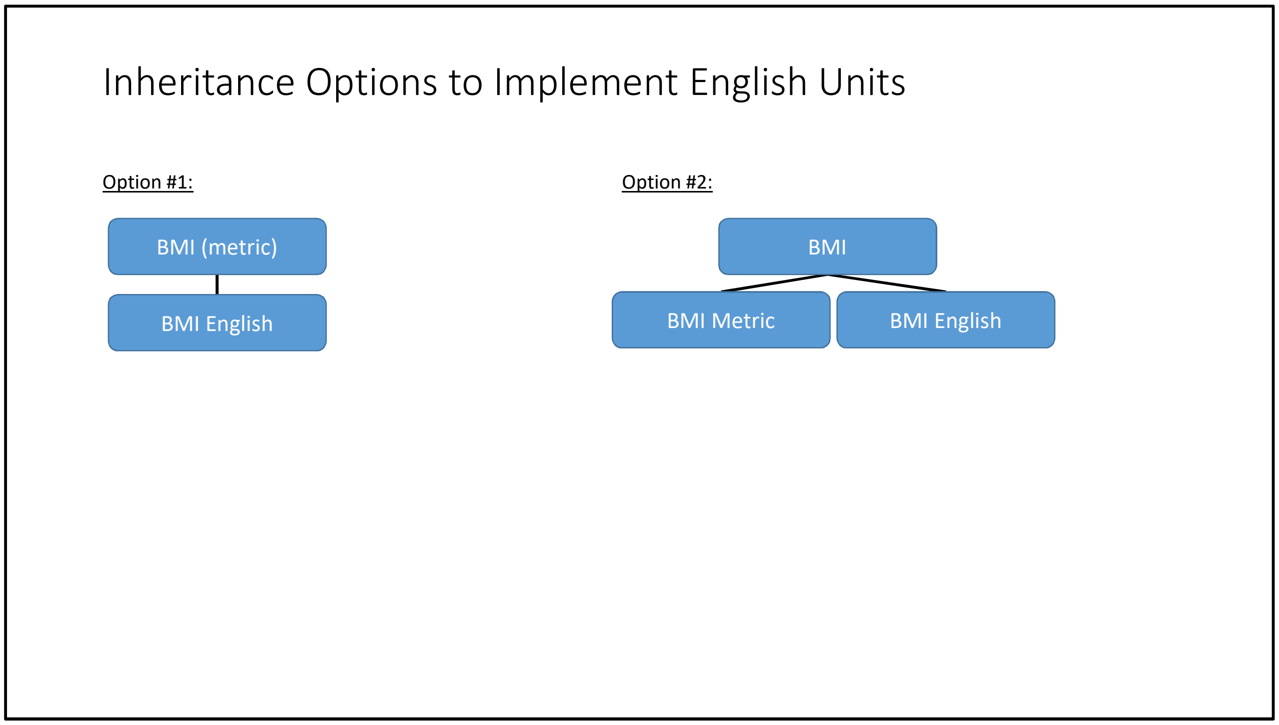
```java
//************************************************************
// BMI Calculator (OOP Java)
// BMI = weight over height squared

class BMI {
    public float CalcBMI(float height, float weight) {
         return weight / (height * height);
    }
}

class CalcBMI {
    public static void main(String[] args) {
        BMI myBMI = new BMI();
        float BMIresult = myBMI.CalcBMI(
            (float)((6.0 * 12.0) + 1.0) /*height*/,
            (float)190.0 /*weight*/);

        System.out.println(BMIresult);
    }
}
```

The Encapsulation concept is not limited to is NOT limited to object-oriented languages and environments. Nearly every language and platform support Encapsulation. In my opinion Encapsulation is fundamental to all development. Object-oriented environments do add some wonderful tools to make Encapsulation elegant.

Consider: How would you add protective code around setting height to 0 in the first option? … in the second encapsulated example? Once again consider reuse, testing, and additional modification in thousands of lines off code.

Encapsulation is a feature of nearly all modern development languages… not just object-oriented languages.

1 – Minimize variable scope: (1)none, (2)local, (3)method parameters, (4)private attribute, (5)protected attribute, and (6)public



Inheritance Options to Implement English Units

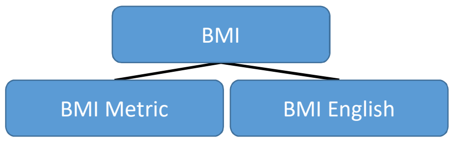
Option #1:

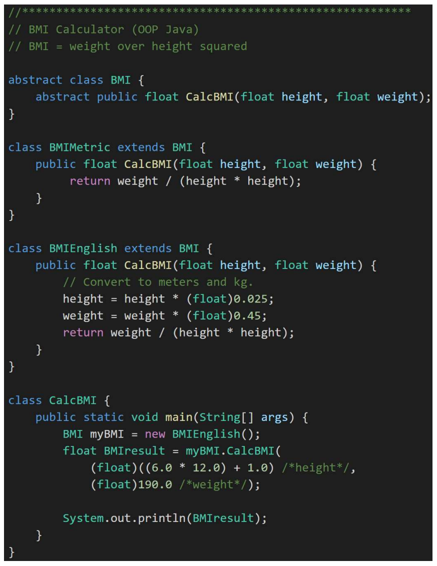
Option #2:

In the real world where not breaking existing code (causing a retest or potentially a defect) is a VERY high priority, we would have chosen Option #1. It is simpler, less risky, more practical, and can be done with less code. As we go through these examples, always try to imagine thousands of lines of code instead of tens of lines of code. Complexity grows exponentially as the number lines of code (and developers) grow.

For our Learning exercise we will implement Option #2. It is a more pure and elegant implementation where BMI Metric and BMI English are at the same level in the class hierarchy… which satisfies my artistic needs. It also will allow us to demonstrate Abstraction, Superclass, and Subclass at the same time we are demonstrating Inheritance.

Inheritance to implement English units… And Abstraction



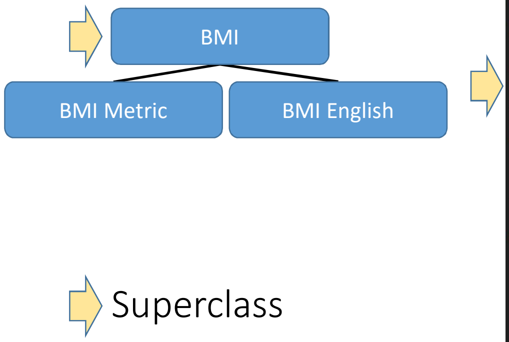


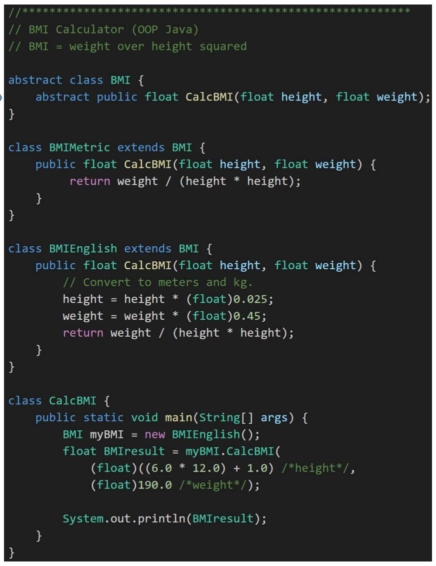
```java
//**********************************************************
// BMI Calculator (OOP Java)
// BMI = weight over height squared

abstract class BMI {
    abstract public float CalcBMI(float height, float weight);
}

class BMIMetric extends BMI {
    public float CalcBMI(float height, float weight) {
         return weight / (height * height);
    }
}

class BMIEnglish extends BMI {
    public float CalcBMI(float height, float weight) {
        // Convert to meters and kg.
        height = height * (float)0.025;
        weight = weight * (float)0.45;
        return weight / (height * height);
    }
}

class CalcBMI {
    public static void main(String[] args) {
        BMI myBMI = new BMIEnglish();
        float BMIresult = myBMI.CalcBMI(
            (float)((6.0 * 12.0) + 1.0) /*height*/,
            (float)190.0 /*weight*/);

        System.out.println(BMIresult);
    }
}
```

Consider: How would you add protective code around setting height to 0 in the first option? … in the second encapsulated example? Once again consider reuse, testing, and additional modification in thousands of lines off code.

Encapsulation is a feature of nearly all modern development languages… not just object-oriented languages.

Identify the superclass and the subclass in an inheritance relationship



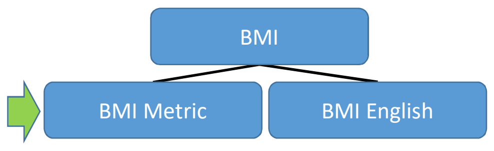


```
//**********************************************************
// BMI Calculator (OOP Java)
// BMI = weight over height squared

abstract class BMI {
    abstract public float CalcBMI(float height, float weight);
}

class BMIMetric extends BMI {
    public float CalcBMI(float height, float weight) {
         return weight / (height * height);
    }
}

class BMIEnglish extends BMI {
    public float CalcBMI(float height, float weight) {
        // Convert to meters and kg.
        height = height * (float)0.025;
        weight = weight * (float)0.45;
        return weight / (height * height);
    }
}

class CalcBMI {
    public static void main(String[] args) {
        BMI myBMI = new BMIEnglish();
        float BMIresult = myBMI.CalcBMI(
            (float)((6.0 * 12.0) + 1.0) /*height*/,
            (float)190.0 /*weight*/);

        System.out.println(BMIresult);
    }
}
```

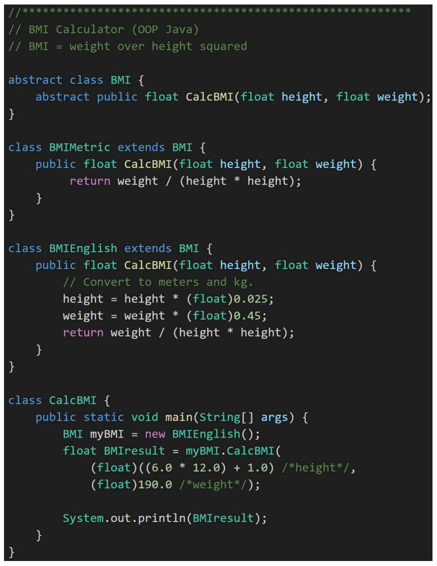
Consider: How would you add protective code around setting height to 0 in the first option? … in the second encapsulated example? Once again consider reuse, testing, and additional modification in thousands of lines off code.

Encapsulation is a feature of nearly all modern development languages… not just object-oriented languages.

Identify the superclass and the subclass in an inheritance relationship



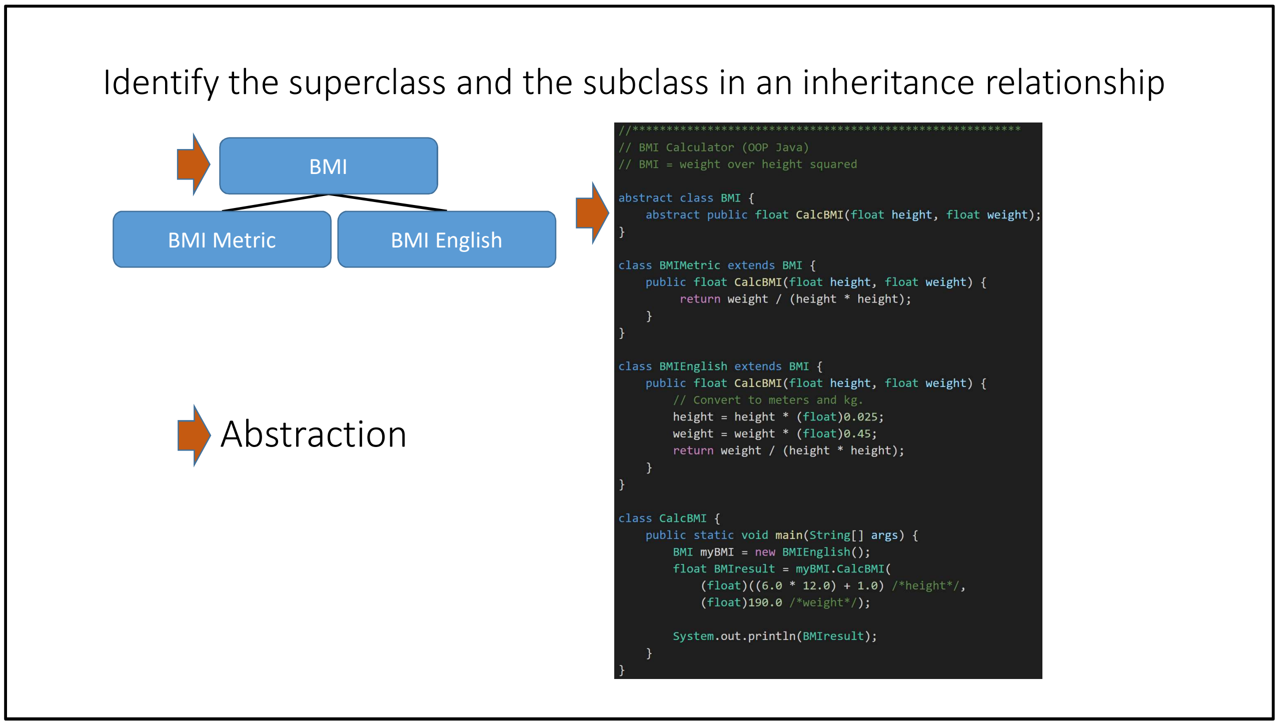
Subclass



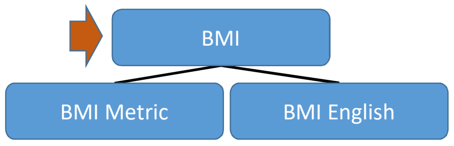
```
//**********************************************************
// BMI Calculator (OOP Java)
// BMI = weight over height squared

abstract class BMI {
    abstract public float CalcBMI(float height, float weight);
}

class BMIMetric extends BMI {
    public float CalcBMI(float height, float weight) {
         return weight / (height * height);
    }
}

class BMIEnglish extends BMI {
    public float CalcBMI(float height, float weight) {
        // Convert to meters and kg.
        height = height * (float)0.025;
        weight = weight * (float)0.45;
        return weight / (height * height);
    }
}

class CalcBMI {
    public static void main(String[] args) {
        BMI myBMI = new BMIEnglish();
        float BMIresult = myBMI.CalcBMI(
            (float)((6.0 * 12.0) + 1.0) /*height*/,
            (float)190.0 /*weight*/);

        System.out.println(BMIresult);
    }
}
```

Consider: How would you add protective code around setting height to 0 in the first option? … in the second encapsulated example? Once again consider reuse, testing, and additional modification in thousands of lines off code.

Encapsulation is a feature of nearly all modern development languages… not just object-oriented languages.



Identify the superclass and the subclass in an inheritance relationship



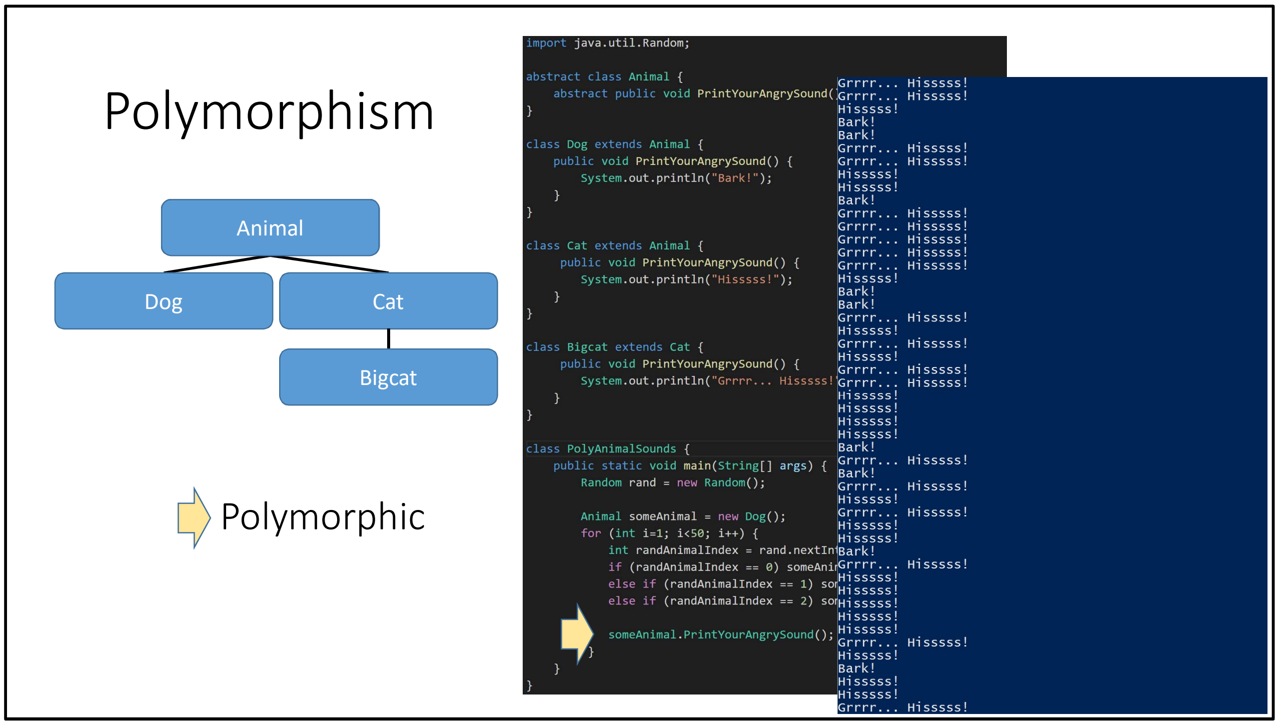
Abstraction

```java
//*********************************************************
// BMI Calculator (OOP Java)
// BMI = weight over height squared

abstract class BMI {
    abstract public float CalcBMI(float height, float weight);
}

class BMIMetric extends BMI {
    public float CalcBMI(float height, float weight) {
         return weight / (height * height);
    }
}

class BMIEnglish extends BMI {
    public float CalcBMI(float height, float weight) {
        // Convert to meters and kg.
        height = height * (float)0.025;
        weight = weight * (float)0.45;
        return weight / (height * height);
    }
}

class CalcBMI {
    public static void main(String[] args) {
        BMI myBMI = new BMIEnglish();
        float BMIresult = myBMI.CalcBMI(
            (float)((6.0 * 12.0) + 1.0) /*height*/,
            (float)190.0 /*weight*/);

        System.out.println(BMIresult);
    }
}
```

Abstraction
Abstraction is another key concept. Something is abstract when it is a concept but is not concrete or defined enough to actually be built. Generally, in OO design, we start with abstract things, and then we build on them through inheritance.

An <u>*abstract class*</u> is one that has one or more *abstract methods*.
An *abstract method* is a method / function that has no body – just a name, return type, and parameters.
An <u>*interface*</u> is the strictest interpretation of an abstract class – it is a data structure that consists entirely of abstract methods. In other words, none of its methods/functions have a body.

Abstraction is related to inheritance. Often, when we construct families of related objects, we start the family with an abstract class that represents the least common denominator for everyone in that family. In other words, what do all classes that are part of that family have in common? We often (not always, but often) put that in an abstract class.



```
import java.util.Random;

abstract class Animal {
    abstract public void PrintYourAngrySound(
}

class Dog extends Animal {
    public void PrintYourAngrySound() {
        System.out.println("Bark!");
    }
}

class Cat extends Animal {
     public void PrintYourAngrySound() {
        System.out.println("Hisssss!");
    }
}

class Bigcat extends Cat {
     public void PrintYourAngrySound() {
        System.out.println("Grrrr... Hisssss!
    }
}

class PolyAnimalSounds {
    public static void main(String[] args) {
        Random rand = new Random();

        Animal someAnimal = new Dog();
        for (int i=1; i<50; i++) {
            int randAnimalIndex = rand.nextInt
            if (randAnimalIndex == 0) someAni
            else if (randAnimalIndex == 1) so
            else if (randAnimalIndex == 2) so

            someAnimal.PrintYourAngrySound();
        }
    }
}
```

```
Grrrr... Hisssss!
Grrrr... Hisssss!
Hisssss!
Bark!
Bark!
Grrrr... Hisssss!
Grrrr... Hisssss!
Hisssss!
Hisssss!
Bark!
Grrrr... Hisssss!
Grrrr... Hisssss!
Grrrr... Hisssss!
Grrrr... Hisssss!
Grrrr... Hisssss!
Hisssss!
Bark!
Bark!
Grrrr... Hisssss!
Hisssss!
Grrrr... Hisssss!
Hisssss!
Grrrr... Hisssss!
Grrrr... Hisssss!
Hisssss!
Hisssss!
Hisssss!
Hisssss!
Bark!
Grrrr... Hisssss!
Bark!
Grrrr... Hisssss!
Hisssss!
Grrrr... Hisssss!
Hisssss!
Hisssss!
Bark!
Grrrr... Hisssss!
Hisssss!
Hisssss!
Hisssss!
Hisssss!
Hisssss!
Grrrr... Hisssss!
Hisssss!
Bark!
Hisssss!
Hisssss!
Grrrr... Hisssss!
```

Notice how "someAnimal" can behave like "Dog", "Cat", or "Bigcat" depending the random number generated. Can you come up with a brief animal based interview example for Polymorphism. Be sure to throw in "oh yes, Polymorphism is usually implemented with virtual methods and a virtual method table" to get full credit at the interview. It may also be good to know what that actually means just in case one of the interviewers really know what a virtual method table is and has a follow up question. Virtual methods are similar to abstract methods. The difference is that Virtual methods CAN be overridden in a Subclass where Abstract methods MUST be overwritten in a Subclass.

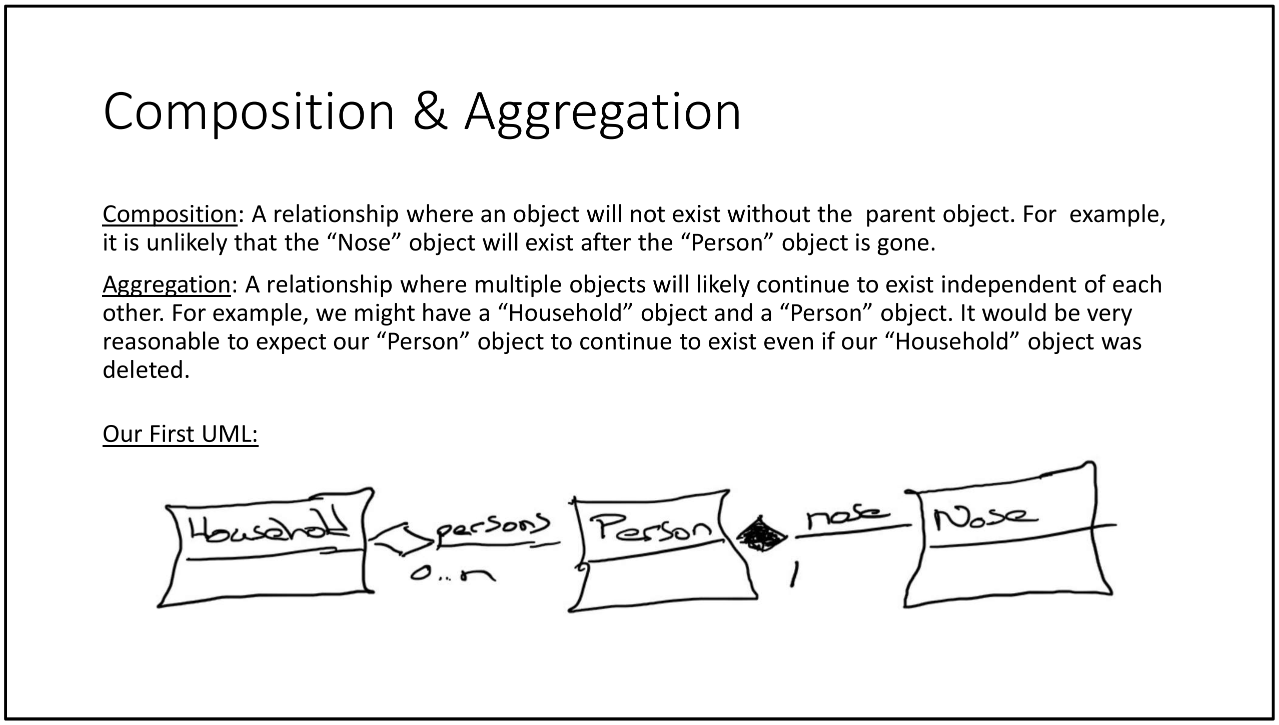
Polymorphism
Polymorphism enables you to process collections of related things generically. This is particularly useful when you want to use a loop to march through a collection of items.

Example of polymorphism: a for loop that moves through the entries of a list. The list might be of a collection of related kinds of object. We can refer to each of the objects in the list through a generic variable (whose data type matches the one that all are ultimately related to). But, when we invoke a particular function that all members of the family share, each will respond by performing that function in their own specific way. For example, we could have a collection of Shape objects. We could refer to each entry in the Shape list through a generic Shape variable, even though the actual entries in the list are specific kinds of shapes – Circle, Rectangle, etc. All Shape objects might have the ability to calculate their own area. When we refer to an object in the list through a generic Shape variable and tell it to calculate its area, thanks to polymorphism, the circle version of the area() function will be called when we're dealing with a circle, and the Rectangle version of area() will be called when we're dealing with a rectangle, etc.

```
Shape[] shapes = new Shape[3];
shapes[0] = new Circle();
shapes[1] = new Rectangle();
shapes[2] = new Triangle();
for (Shape s : shapes) {
  System.out.println(s.area());
}
```

The first time through the for loop, we'll call the Circle.area() function – actually, we won't; it will happen automatically. The next time through, we'll call the Rectangle version, and then we'll call the Triangle version.

Polymorphism is implemented behind the scenes using a Virtual Method Table (VMT). The VMT keeps track of where various related classes' same-named functions are located in memory. Using the VMT, the operating system is able to figure out which code to implement when we tell each shape to fire its area() function.

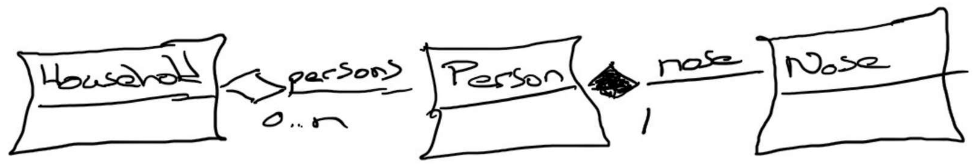


Composition & Aggregation

Composition: A relationship where an object will not exist without the parent object. For example, it is unlikely that the “Nose” object will exist after the “Person” object is gone.

Aggregation: A relationship where multiple objects will likely continue to exist independent of each other. For example, we might have a “Household” object and a “Person” object. It would be very reasonable to expect our “Person” object to continue to exist even if our “Household” object was deleted.

Our First UML:



Composition

An example of composition: the relationship between Face and Nose, Mouth, and Eye

```
class Face {
        Nose n;
        Mouth m;
        Eye le;
        Eye re;
}
```

We probably wouldn’t let the Nose, Mouth, or Eye objects live beyond the Face. They are owned exclusively by the Face. That’s what composition is: exclusive ownership.

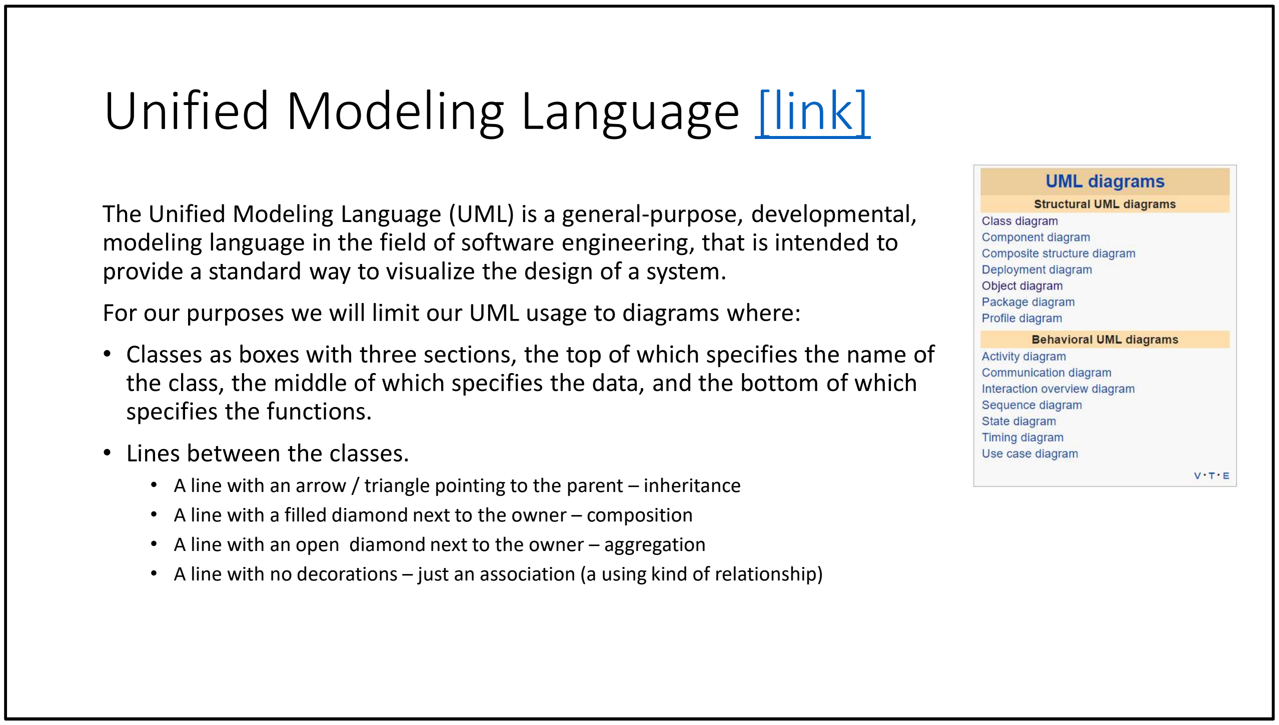
One clear sign that we are dealing with composition is if the owner (Face, for example) is responsible for actually creating the objects it owns. Then, clearly, the things that are owned – the Nose, Mouth, etc. – could not have existed on their own and are therefore exclusively owned by the owner object.

Aggregation

Aggregation is also a form of ownership, but it’s non-exclusive ownership. The owned objects can live on and perhaps existed prior to the owned object.

An example of aggregation: A library patron borrows a book. That’s not exclusive ownership, since several people can borrow a book over its lifetime.

The easy way to tell if something is composition or aggregation is to ask if the owner is responsible for creating and destroying the thing that is owned. If so, it’s a composition relationship.

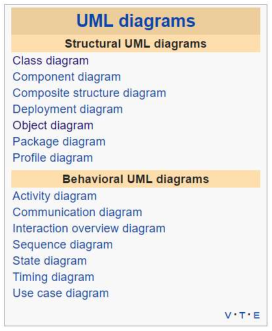


Unified Modeling Language [link]

The Unified Modeling Language (UML) is a general-purpose, developmental, modeling language in the field of software engineering, that is intended to provide a standard way to visualize the design of a system.

For our purposes we will limit our UML usage to diagrams where:

- Classes as boxes with three sections, the top of which specifies the name of the class, the middle of which specifies the data, and the bottom of which specifies the functions.
- Lines between the classes.
  - A line with an arrow / triangle pointing to the parent – inheritance
  - A line with a filled diamond next to the owner – composition
  - A line with an open diamond next to the owner – aggregation
  - A line with no decorations – just an association (a using kind of relationship)



UML diagrams

Structural UML diagrams

Class diagram
Component diagram
Composite structure diagram
Deployment diagram
Object diagram
Package diagram
Profile diagram

Behavioral UML diagrams

Activity diagram
Communication diagram
Interaction overview diagram
Sequence diagram
State diagram
Timing diagram
Use case diagram

V·T·E

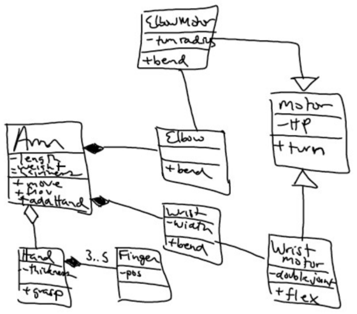
UML can describe many different things and become quite complex. There is some debate, particularly in the Agile development methodology circles, as to the benefits of this documentation outweigh the cost. Most developers agree that at least a basic understanding of UML diagrams aids greatly in the impromptu “chalkboard” design discussions that occur constantly during the design, implementation, and testing of a project.

UML helps us see the design without having to wade through lots of text to understand it.

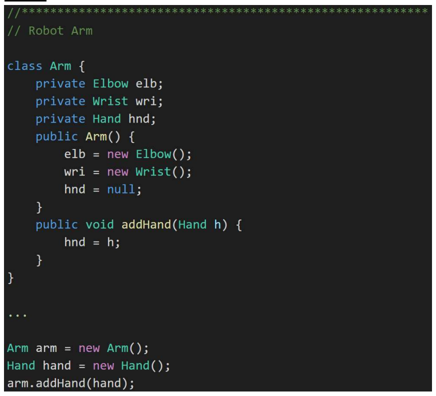
UML reached its peak with Iterative Development methodologies like the Rational Unified Process (RUP) and OpenUp. It still is viewed as a critical part of Design for many Waterfall and Iterative organizations.

UML Example: Robot Arm

UML:



Java:



```java
//**********************************************************
// Robot Arm

class Arm {
    private Elbow elb;
    private Wrist wri;
    private Hand hnd;
    public Arm() {
        elb = new Elbow();
        wri = new Wrist();
        hnd = null;
    }
    public void addHand(Hand h) {
        hnd = h;
    }
}

...

Arm arm = new Arm();
Hand hand = new Hand();
arm.addHand(hand);
```

UML can describe many different things and become quite complex. There is some debate, particularly in the Agile development methodology circles, as to the benefits of this documentation outweigh the cost. Most developers agree that at least a basic understanding of UML diagrams aids greatly in the impromptu “chalkboard” design discussions that occur constantly during the design, implementation, and testing of a project.

UML helps us see the design without having to wade through lots of text to understand it.

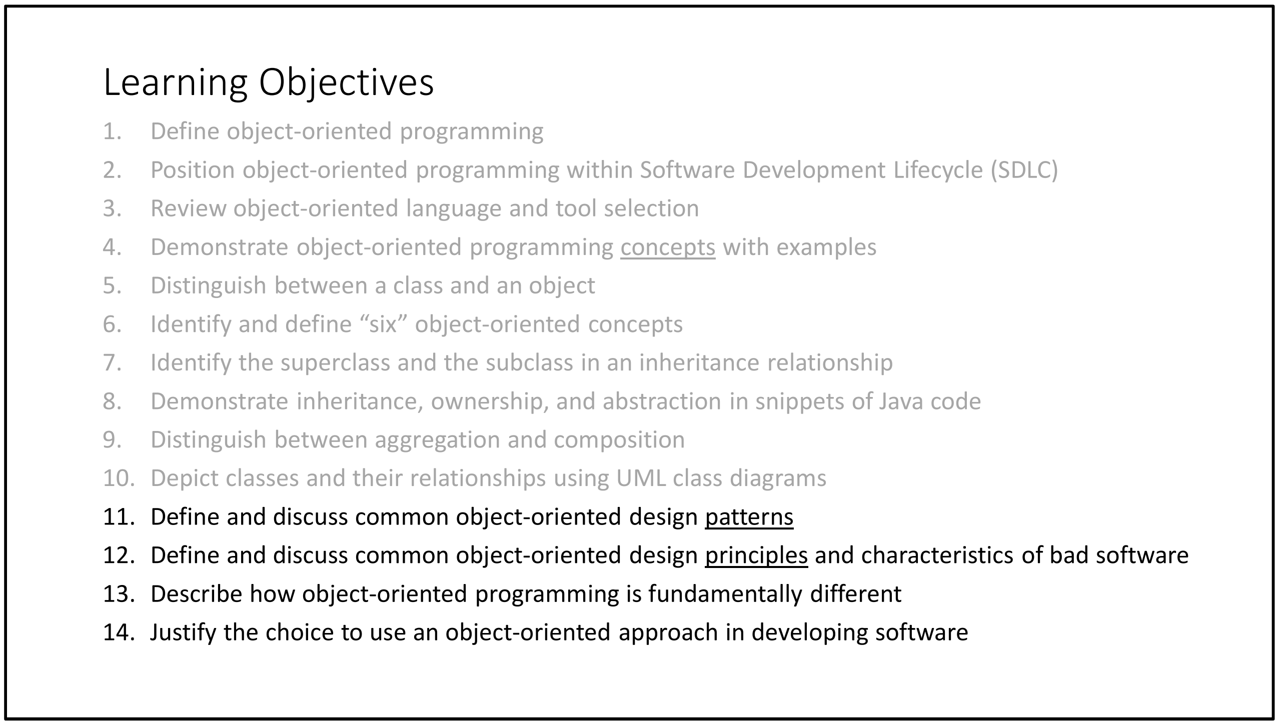
UML
UML, or Unified Modeling Language, is a way to show a system’s architecture in graphical form. UML represents
classes as boxes with three sections, the top of which specifies the name of the class, the middle of which specifies the data, and the bottom of which specifies the functions.
Lines between the classes.

A line with an arrow / triangle pointing to the parent – inheritance

A line with a filled diamond next to the owner – composition

    A line with an open diamond next to the owner – aggregation
    A line with no decorations – just an association (a using kind of relationship)

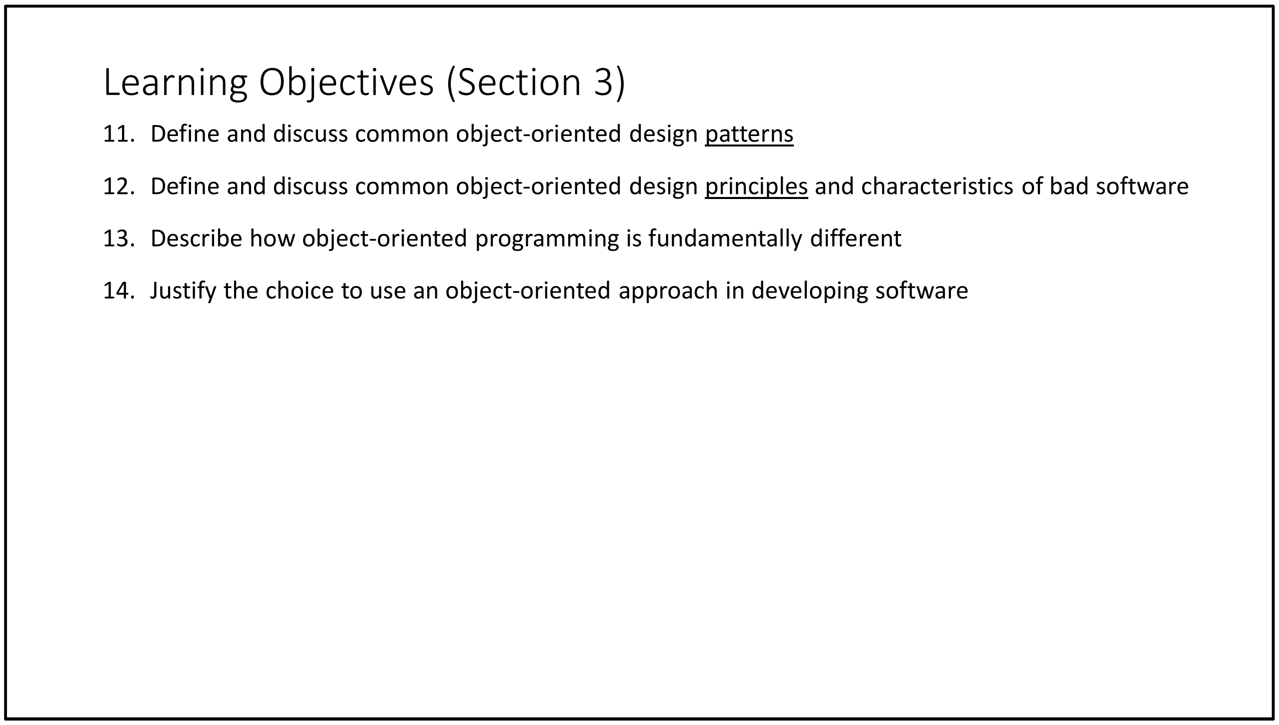
UML helps us see the design without having to wade through lots of text to understand it.



Learning Objectives

1. Define object-oriented programming
2. Position object-oriented programming within Software Development Lifecycle (SDLC)
3. Review object-oriented language and tool selection
4. Demonstrate object-oriented programming concepts with examples
5. Distinguish between a class and an object
6. Identify and define “six” object-oriented concepts
7. Identify the superclass and the subclass in an inheritance relationship
8. Demonstrate inheritance, ownership, and abstraction in snippets of Java code
9. Distinguish between aggregation and composition
10. Depict classes and their relationships using UML class diagrams
11. Define and discuss common object-oriented design patterns
12. Define and discuss common object-oriented design principles and characteristics of bad software
13. Describe how object-oriented programming is fundamentally different
14. Justify the choice to use an object-oriented approach in developing software

You will notice very quickly that I prefer examples and actual source code over more philosophical discussion. Please feel free to ask for more development philosophy and background if you desire more.



Learning Objectives (Section 3)

11. Define and discuss common object-oriented design patterns
12. Define and discuss common object-oriented design principles and characteristics of bad software
13. Describe how object-oriented programming is fundamentally different
14. Justify the choice to use an object-oriented approach in developing software

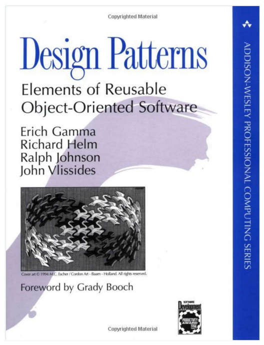
You will notice very quickly that I prefer examples and actual source code over more philosophical discussion. Please feel free to ask for more development philosophy and background if you desire more.

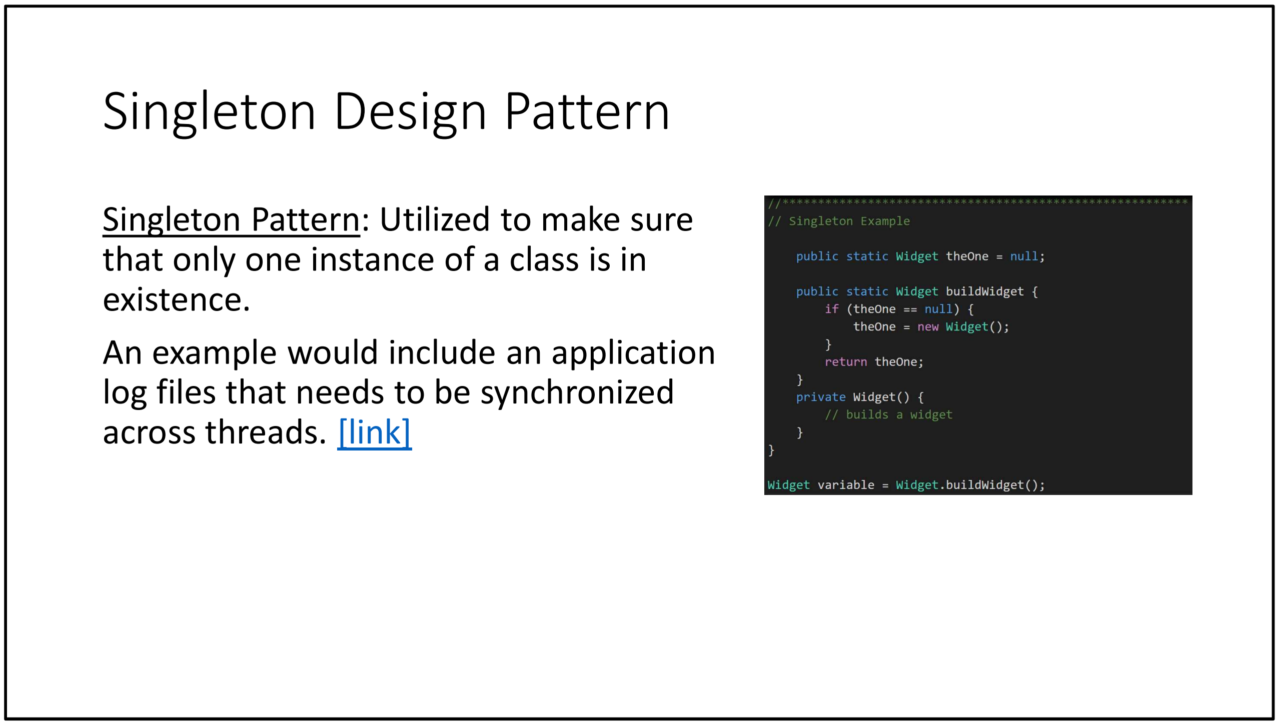
Object-Oriented Design Patterns

Definition: A software design pattern is a commonly repeated approach to constructing software. These approaches are commonly repeated because they have found to be useful.

Common Patterns Include:

- Singleton
- Factory
- Delegation
- Model-View-Controller
- Others


Design Patterns
Elements of Reusable Object-Oriented Software
Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides
Foreword by Grady Booch
ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES



Singleton Design Pattern

Singleton Pattern: Utilized to make sure that only one instance of a class is in existence.

An example would include an application log files that needs to be synchronized across threads. [link]

```
//*********************************************************
// Singleton Example

    public static Widget theOne = null;

    public static Widget buildWidget {
        if (theOne == null) {
            theOne = new Widget();
        }
        return theOne;
    }
    private Widget() {
        // builds a widget
    }
}

Widget variable = Widget.buildWidget();
```

Singleton (making sure there is just one instance of something to avoid conflicts - http://www.oodesign.com/singleton-pattern.html) These are great for coordinating activity across multiple threads of execution.

A singleton can be written by making the constructor for the class private and equipping the class with a static function that ensures that the constructor is called only if no other objects of that class already exist.

Singleton Example

```
class Widget {

            public static Widget theOne = null;

            public static Widget buildWidget {
                        if (theOne == null) {
                                    theOne = new Widget();
                        }
                        return theOne;
```

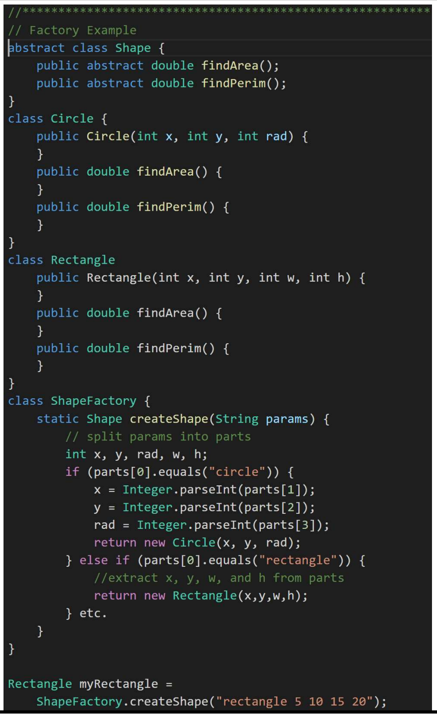
```
        }
        private Widget() {
                // builds a widget
        }
}

Widget variable = Widget.buildWidget();
```

Factory Design Pattern

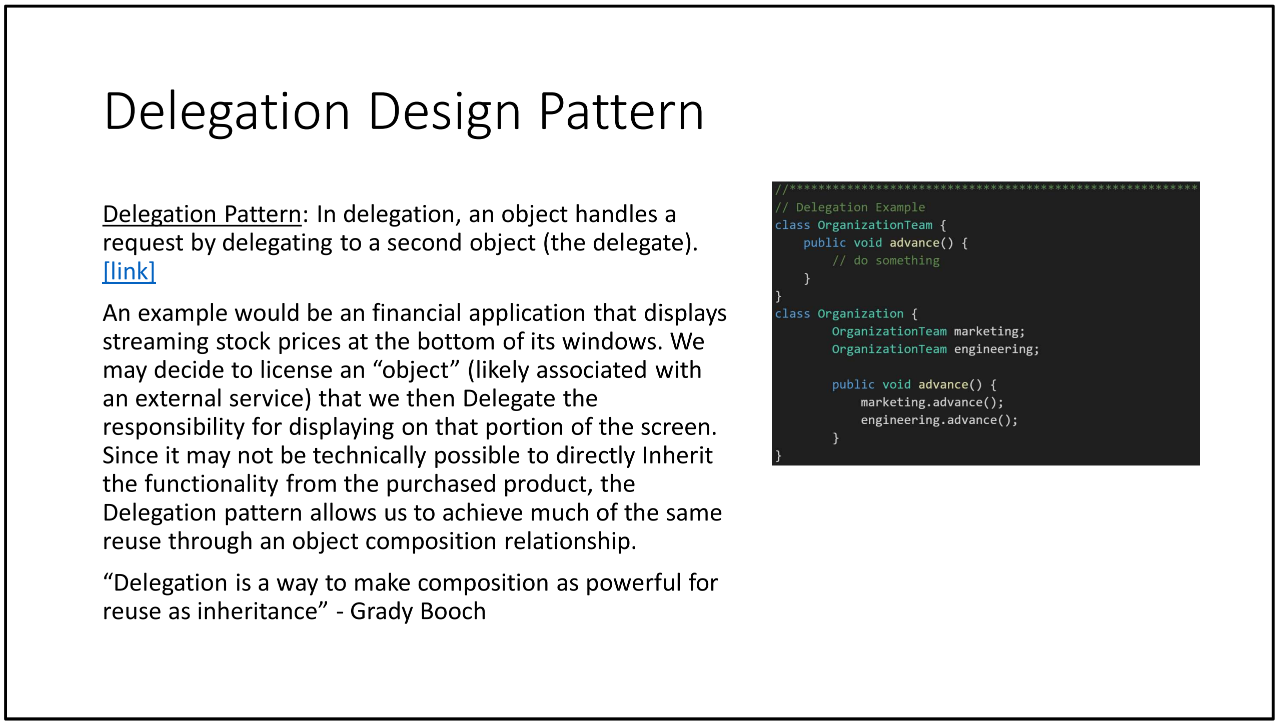
Factory Pattern: Utilized to create an object without exposing how it is created. [link]

An example would have been in our Polymorphic Animal. We could have made an Animal Factory that his the fact that we were just creating random Dogs, Cats, and Bigcats. Then if we wanted to change the ratios or add Animals, we could do it without forcing code changes outside of the Animal Factory.

Another example would be a Shape Factory that return shapes based on input, but does not expose how the shapes are created. See coding example.



```
//**********************************************************
// Factory Example
abstract class Shape {
    public abstract double findArea();
    public abstract double findPerim();
}
class Circle {
    public Circle(int x, int y, int rad) {
    }
    public double findArea() {
    }
    public double findPerim() {
    }
}
class Rectangle
    public Rectangle(int x, int y, int w, int h) {
    }
    public double findArea() {
    }
    public double findPerim() {
    }
}
class ShapeFactory {
    static Shape createShape(String params) {
        // split params into parts
        int x, y, rad, w, h;
        if (parts[0].equals("circle")) {
            x = Integer.parseInt(parts[1]);
            y = Integer.parseInt(parts[2]);
            rad = Integer.parseInt(parts[3]);
            return new Circle(x, y, rad);
        } else if (parts[0].equals("rectangle")) {
            //extract x, y, w, and h from parts
            return new Rectangle(x,y,w,h);
        } etc.
    }
}

Rectangle myRectangle =
    ShapeFactory.createShape("rectangle 5 10 15 20");
```



Delegation Design Pattern

Delegation Pattern: In delegation, an object handles a request by delegating to a second object (the delegate). [link]

An example would be an financial application that displays streaming stock prices at the bottom of its windows. We may decide to license an "object" (likely associated with an external service) that we then Delegate the responsibility for displaying on that portion of the screen. Since it may not be technically possible to directly Inherit the functionality from the purchased product, the Delegation pattern allows us to achieve much of the same reuse through an object composition relationship.

"Delegation is a way to make composition as powerful for reuse as inheritance" - Grady Booch

```
//********************************************************
// Delegation Example
class OrganizationTeam {
    public void advance() {
        // do something
    }
}
class Organization {
        OrganizationTeam marketing;
        OrganizationTeam engineering;

        public void advance() {
            marketing.advance();
            engineering.advance();
        }
}
```

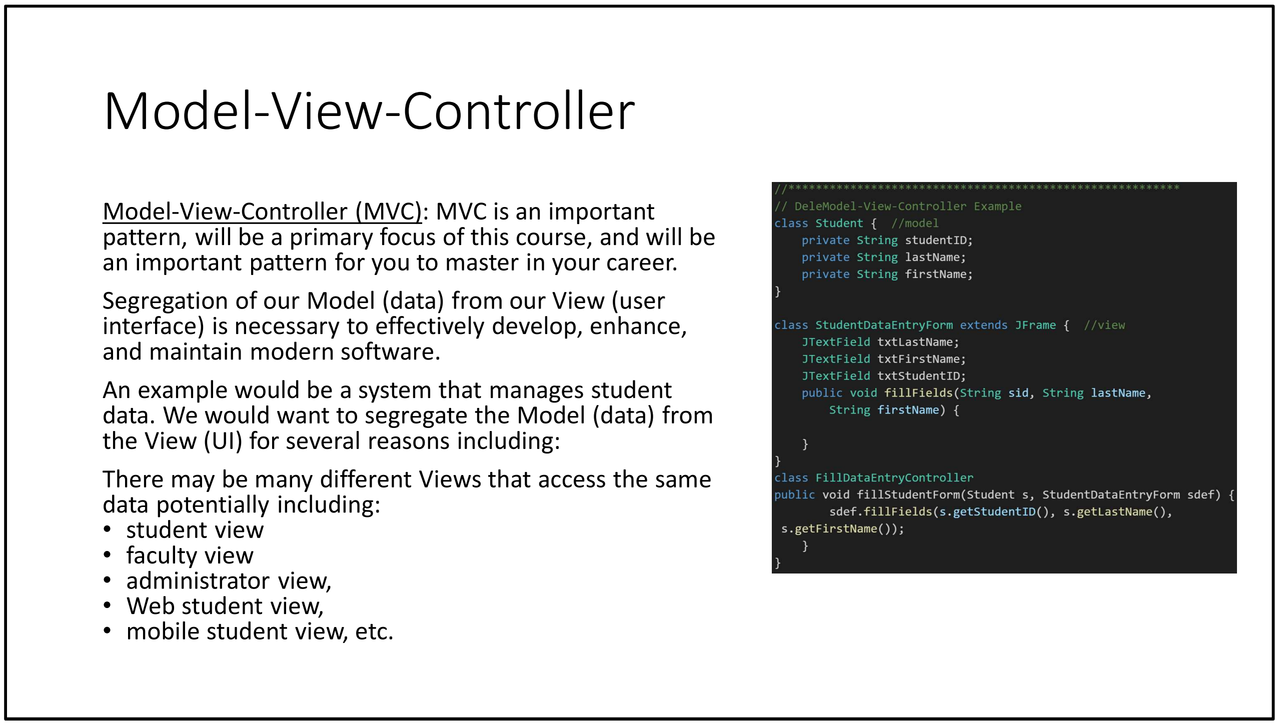
Delegation (chain of command; ask an object to do something, which tells something else to do that thing)

Example:

```
class OrganizationTeam {
        public void advance() {
                // do something
        }
}
class Organization {
                OrganizationTeam marketing;
                OrganizationTeam engineering;

                public void advance() {
                        marketing.advance();
                        engineering.advance();
                }
}
```

Another Example of the Delegation pattern
In this example, we have software for managing inventory for a manufacturer who makes classroom furniture. We have a variety of writing surfaces that consist of parts. We want to print out our inventory of parts.

```
class Part {

}
class WritingSurface {
        private Part[] parts;
        public Part[] listParts() {

        }
}
class Podium extends WritingSurface {

}
class StudentDesk extends WritingSurface {

}
class PartsInventory {
        // list all the components that comprise a factory's models
        WritingSurface[] models;
        public Part[] listParts() {
                for (WritingSurface ws : models) {
                        ws.listParts();
                }
        }
}

PartsInventory pi = new PartsInventory();
pi.listParts();
```



Model-View-Controller

Model-View-Controller (MVC): MVC is an important pattern, will be a primary focus of this course, and will be an important pattern for you to master in your career.

Segregation of our Model (data) from our View (user interface) is necessary to effectively develop, enhance, and maintain modern software.

An example would be a system that manages student data. We would want to segregate the Model (data) from the View (UI) for several reasons including:

There may be many different Views that access the same data potentially including:
- student view
- faculty view
- administrator view,
- Web student view,
- mobile student view, etc.

```
//*********************************************************
// DeleModel-View-Controller Example
class Student {  //model
    private String studentID;
    private String lastName;
    private String firstName;
}

class StudentDataEntryForm extends JFrame {  //view
    JTextField txtLastName;
    JTextField txtFirstName;
    JTextField txtStudentID;
    public void fillFields(String sid, String lastName,
        String firstName) {

    }
}
class FillDataEntryController
public void fillStudentForm(Student s, StudentDataEntryForm sdef) {
        sdef.fillFields(s.getStudentID(), s.getLastName(),
 s.getFirstName());
    }
}
```

Evolution of UI and Data segregation
- Document-View (View was responsible for View-Controller functionality)
- Model-View-Controller
- Model–View–Viewmodel

Why Use Patterns?
Using patterns helps us write good software more regularly, because they are tried-and-true approaches to writing it.

Each new level of abstraction adds complexity (and likely negatively impacts performance), and should be justified based on the additional value it provides.

Model-View-Controller – our primary focus this semester. It guides us to separate model – the data – from how we view the data – the view – using an intermediary called the controller.

Example of Model-View-Controller: Suppose we have a system that manages student data. The Student class would be the model; a StudentEntryForm could be the view, and a FillStudentEntryFormController could be the controller class responsible for populating a student data entry form with data.

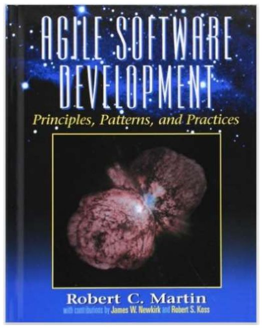
```
class Student {  //model
                    private String studentID;
                    private String lastName;
                    private String firstName;
}

class StudentDataEntryForm extends JFrame {  //view
                    JTextField txtLastName;
                    JTextField txtFirstName;
                    JTextField txtStudentID;
                    public void fillFields(String sid, String lastName, String firstName) {

                    }
}
class FillDataEntryController
public void fillStudentForm(Student s, StudentDataEntryForm sdef) {
                                        sdef.fillFields(s.getStudentID(), s.getLastName(),
 s.getFirstName());
                    }

}
```

Object-Oriented Design Principles

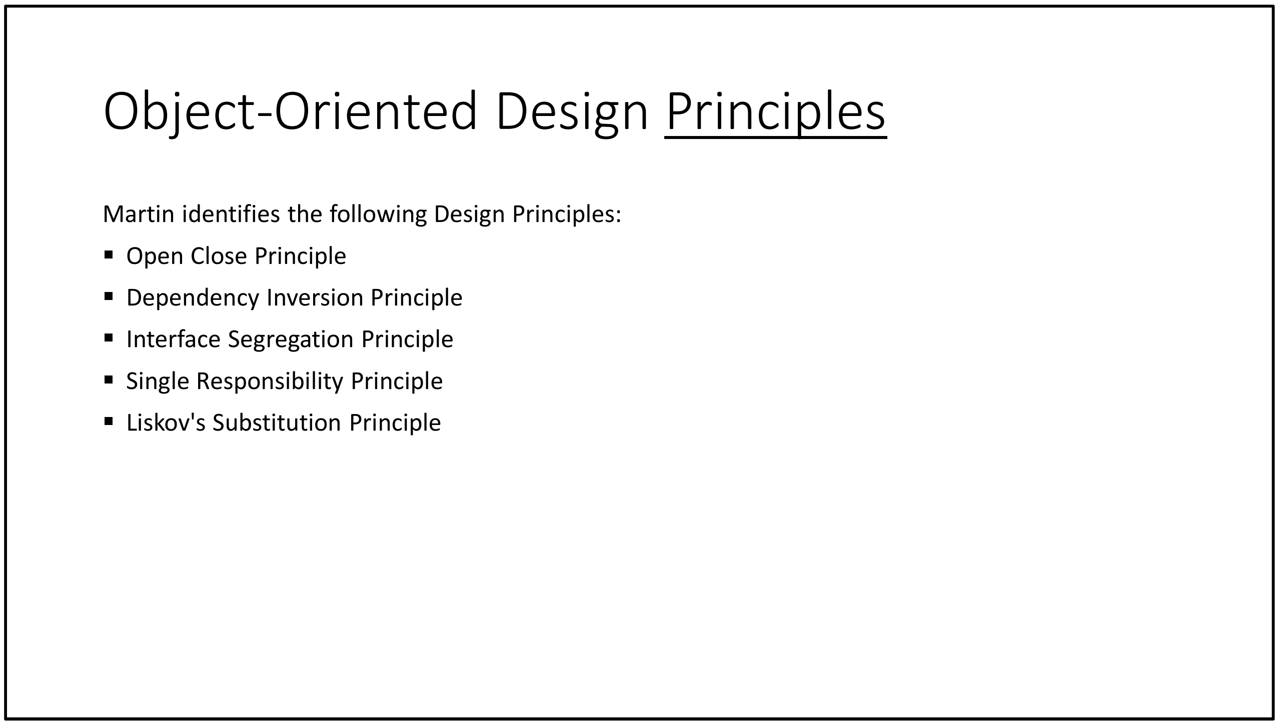
Software design principles represent a set of guidelines that helps us to avoid having a bad design. The design principles are associated to Robert Martin who gathered them in "Agile Software Development: Principles, Patterns, and Practices".

According to Robert Martin there are 3 important characteristics of a bad design that should be avoided:

- Rigidity - It is hard to change because every change affects too many other parts of the system.
- Fragility - When you make a change, unexpected parts of the system break.
- Immobility - It is hard to reuse in another application because it cannot be disentangled from the current application.


Agile Software Development
Principles, Patterns, and Practices
Robert C. Martin
with contributions by James W. Newkirk and Robert S. Koss

Our procedural (C) implementation of BMI would be a great example of Immobility.



Object-Oriented Design Principles

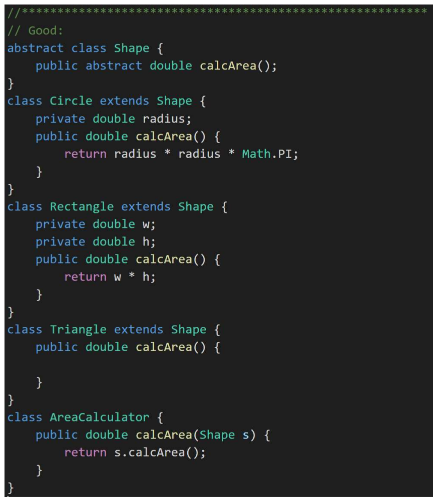
Martin identifies the following Design Principles:

- Open Close Principle
- Dependency Inversion Principle
- Interface Segregation Principle
- Single Responsibility Principle
- Liskov's Substitution Principle

Our procedural (C) implementation of BMI would be a great example of Immobility.

Open Close Principle

Software entities like classes should be open for extension but closed for modifications.



```
//*********************************************************
// Good:
abstract class Shape {
    public abstract double calcArea();
}
class Circle extends Shape {
    private double radius;
    public double calcArea() {
        return radius * radius * Math.PI;
    }
}
class Rectangle extends Shape {
    private double w;
    private double h;
    public double calcArea() {
        return w * h;
    }
}
class Triangle extends Shape {
    public double calcArea() {

    }
}
class AreaCalculator {
    public double calcArea(Shape s) {
        return s.calcArea();
    }
}
```

<u>**Open-Closed Principle Example**</u>

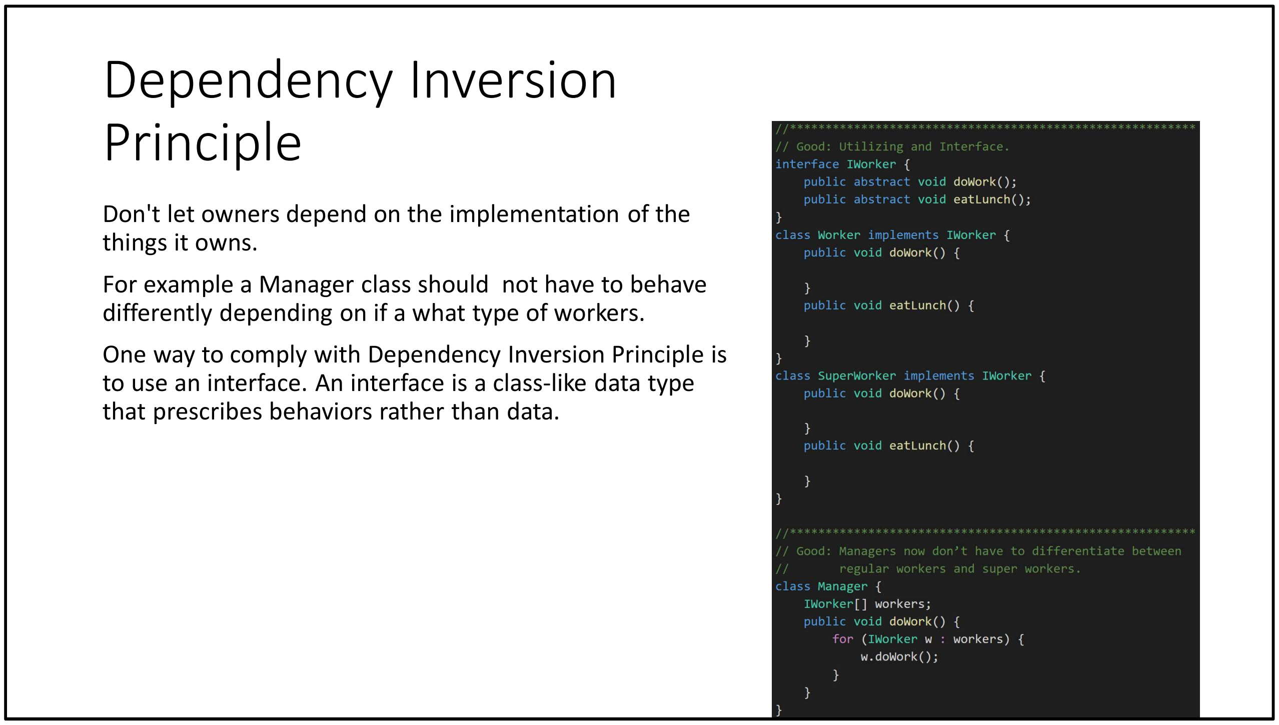
Software should open to extension and closed to modification. You shouldn't have to change the code to have it handle new, related circumstances.

Bad:

```
class AreaCalculator {
            public double calcArea(Shape s) {
                        if (s instanceof Circle) {
                                    Circle c = (Circle)s;
                                    return c.radius * c.radius * Math.PI;
                        } else if (s instanceof Rectangle) {
                                    Rectangle r = (Rectangle)s;
                                    return r.width * r.height;
                        } else if (s instanceof Triangle) {
                                    Triangle t = (Triangle)s;
                                    return 0.5 * t.base * t.height;
                        }
            }
}
```

Good:

```
abstract class Shape {
		public abstract double calcArea();
}
class Circle extends Shape {
		private double radius;
		public double calcArea() {
				return radius * radius * Math.PI;
		}
}
class Rectangle extends Shape {
		private double w;
		private double h;
		public double calcArea() {
				return w * h;
		}
}
class Triangle extends Shape {
		public double calcArea() {

		}
}
class AreaCalculator {
		public double calcArea(Shape s) {
				return s.calcArea();
		}
}
```



Dependency Inversion Principle

Don't let owners depend on the implementation of the things it owns.

For example a Manager class should not have to behave differently depending on if a what type of workers.

One way to comply with Dependency Inversion Principle is to use an interface. An interface is a class-like data type that prescribes behaviors rather than data.

```
//**********************************************************
// Good: Utilizing and Interface.
interface IWorker {
    public abstract void doWork();
    public abstract void eatLunch();
}
class Worker implements IWorker {
    public void doWork() {

    }
    public void eatLunch() {

    }
}
class SuperWorker implements IWorker {
    public void doWork() {

    }
    public void eatLunch() {

    }
}

//**********************************************************
// Good: Managers now don't have to differentiate between
//       regular workers and super workers.
class Manager {
    IWorker[] workers;
    public void doWork() {
        for (IWorker w : workers) {
            w.doWork();
        }
    }
}
```

Dependency Inversion Principle

Don't let owners depend on the implementation of the things it owns.
Example: Manager and Worker

Bad: Managers have to distinguish and handle separately two different kinds of workers

```
class Manager {
		Worker[] workers;
		SuperWorker[] sworkers;
		public void doWork() {
				for (Worker w : workers) {
						w.doWork();
				}
				for (SuperWorker sw : sworkers) {
						sw.doWork();
				}
		}
}
```

One way to comply with Dependency Inversion Principle is to use an **interface**.
An interface is a class-like data type that prescribes behaviors rather than data.

An interface a set of abstract functions.
Classes then *implement* that interface. In other words, they implement all of the function that make up the interface.

Whereas you can you extend only one class, you can implement as many interfaces as you need.

```
interface IWorker {
        public abstract void doWork();
        public abstract void eatLunch();
}
class Worker implements IWorker {
        public void doWork() {

        }
        public void eatLunch() {

        }
}
class SuperWorker implements IWorker {
        public void doWork() {

        }
        public void eatLunch() {

        }
}
```

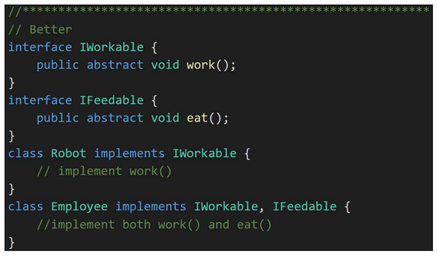
Good: Managers now don’t have to differentiate between regular workers and super workers.

```
class Manager {
        IWorker[] workers;
        public void doWork() {
                for (IWorker w : workers) {
                        w.doWork();
                }
        }
}
```

Interface Segregation Principle

Clients should not be forced to depend upon interfaces that they don't use.

The previous example has a minor flaw. Consider the IWorker interface. It specifies that workers both eat lunch and do work. What if we end up building and using robotic workers? They don't have to eat. So, our definition of Worker includes too much and therefore can only be clumsily applied to situations where our understanding of what a worker is might change.



```
//*********************************************************
// Better
interface IWorkable {
    public abstract void work();
}
interface IFeedable {
    public abstract void eat();
}
class Robot implements IWorkable {
    // implement work()
}
class Employee implements IWorkable, IFeedable {
    //implement both work() and eat()
}
```

<u>Interface Segregation Principle</u>

The example we just did has a teensy flaw. Consider the IWorker interface. It specifies that workers both eat lunch and do work. What if we end up building and using robotic workers? They don't have to eat. So, our definition of Worker includes too much and therefore can only be clumsily applied to situations where our understanding of what a worker is might change.

NOT IDEAL:

```
interface IWorker {
                public abstract void eat();
                public abstract void work();
}

public class Robot implements IWorker {
                // implement work() and, somehow, eat()
}
```

BETTER:

```
interface IWorkable {
```

```
                public abstract void work();
}
interface IFeedable {
                public abstract void eat();
}
class Robot implements IWorkable {
                // implement work()
}
class Employee implements IWorkable, IFeedable {
                //implement both work() and eat()
}
```

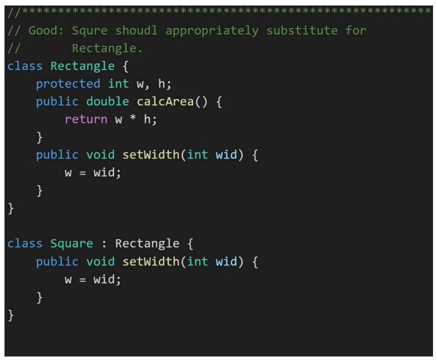
Single Responsibility Principle

A class should have only one reason to change.

An example might be a video player. If you build into the video player class both the responsibility to render the content and control the networking and buffering, you now have two reasons to have to change that VideoPlayer class. Instead, separate out the rendering and the networking/buffering among classes that the VideoPlayer class owns.

Liskov Substitution Principle

Derived classes must be completely substitutable for their base types. And similarly Subclasses should not break core functionality introduced by their parent.



```
//*********************************************************
// Good: Squre shoudl appropriately substitute for
//       Rectangle.
class Rectangle {
    protected int w, h;
    public double calcArea() {
        return w * h;
    }
    public void setWidth(int wid) {
        w = wid;
    }
}

class Square : Rectangle {
    public void setWidth(int wid) {
        w = wid;
    }
}
```

Liskov Substitution Principle

Subclasses should not break core functionality introduced by their parent.

For example: this is bad, because Square is not just setting its width in the setWidth function; it is also setting its height:

```
class Rectangle {
        protected int w, h;
        public double calcArea() {
                return w * h;
        }
        public void setWidth(int wid) {
                w = wid;
        }
}

class Square : Rectangle {
        public void setWidth(int wid, int ht) {
                w = wid;
                h = ht;
```

```
                    }
          }
```

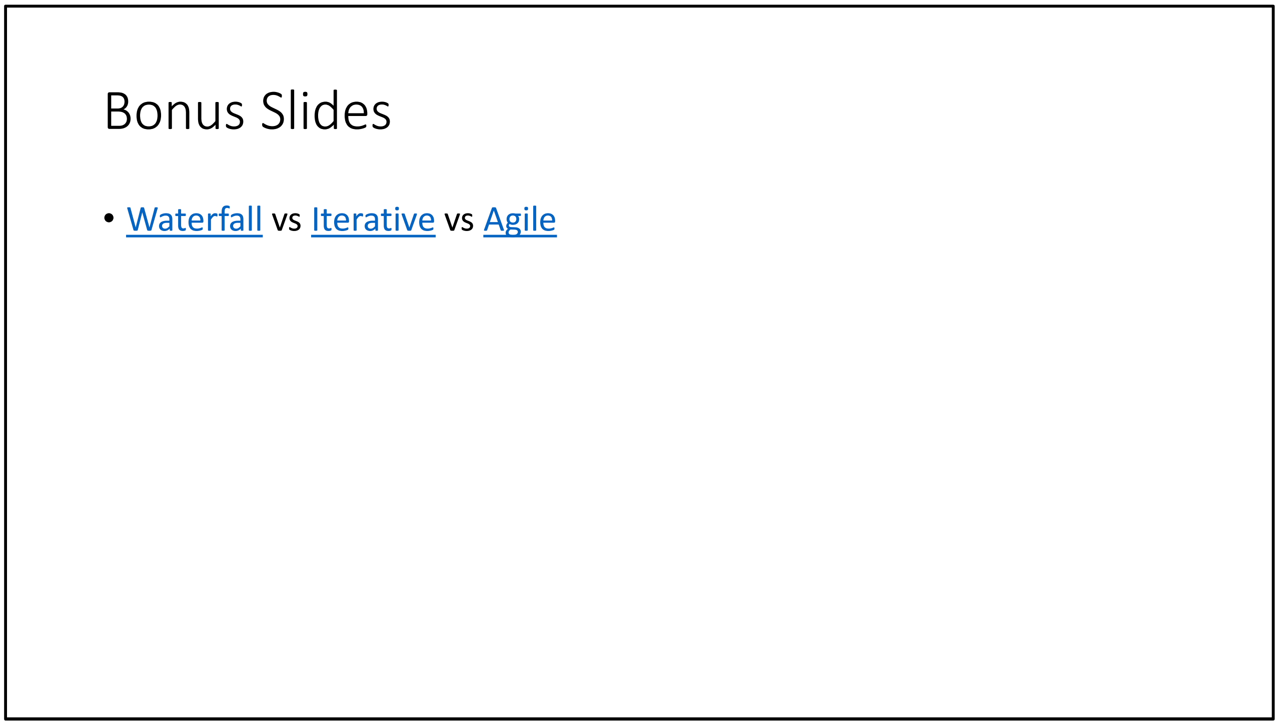
Not cool!

Bottom Line About These Principles
We'll see these principles again as the course continues.

This course mixes theory and practice
We will learn and re-learn these concepts as we learn three object-oriented programming languages: Java and C#

Describe how object-oriented programming is fundamentally different

Justify the choice to use an object-oriented approach in developing software



Bonus Slides

- Waterfall vs Iterative vs Agile

Waterfall vs Iterative vs Agile

	Waterfall	Iterative	Agile
References	United States Department of Defense: DOD-STD-2167A (1985)	Rational Unified Process (RUP) Open Unified Process	Scrum Kanban Scaled Agile Framework (SAFe)
Priorities	Planning and predictability	Architecture, modeling, and efficiency through early detection & fixing of issues	Responsiveness to feedback, efficiency through engineering practices, early detection & fixing of issues
Principles	Execute phases sequentially: 1. Requirements 2. Analysis 3. Design 4. Coding 5. Testing 6. and Operations Define and commit to Scope, Cost, and Timeline "early" Implement strict Change Control	Develop and test iteratively Manage requirements Use components Model visually Verify quality Control changes	Develop, test, deploy, and release iteratively Capture lightweight near term requirements Empower teams Allow requirements to evolve but maintain fixed timelines Apply engineering practices and systems thinking (e.g. TDD) Integrate early user feedback into remaining plan Maintain a collaborative approach between all stakeholders

Waterfall vs Iterative vs Agile (continued)

	Waterfall	Iterative	Agile
Engineering Standards	None specified	Components and Modeling (UML) XP referenced periodically	XP DevOps
Optimization	Specialization & standardization based on skills, location, and/or technology	Architecture, modeling, and iterations	Fast customer feedback, small cross-functional teams, and working software
Planning Horizon Focus	Years – High Months – High Days – Low	Years – Medium Months – Medium Days – Medium	Years – Low Months – Medium Days – High
Scope and Requirements Management	Scope locked-in after Requirements Phase (approximately one-third through the project)	Requirements are stabilized during Elaboration Phase with smaller changes in later phases	Changes in requirements embraced utilizing lowest priority items as Scope buffer
Prevalent Issues	Timeline and cost overruns	Timeline and cost overruns	Reduced functionality release
Industry Success Rates	Low to Medium	Medium	Medium to High

Waterfall vs Iterative vs Agile (continued)

Optional

	Waterfall	Iterative	Agile
Delivery Ownership	Ownership changes from role to role by phase with Project Manager being responsible for overall delivery	Project Manager responsible for completing iteration	Team centric with team responsible for delivery of working feature
Resource Utilization	Role and skill centric with tasks often being restricted to an individual or group	Role and skill centric with tasks often being restricted to an individual or group	Team centric with team members switching roles regularly
Status Reporting	Work progress (% complete) relative to plan	Work progress plus product features delivered by Iteration during Implementation	Product features demonstrated by Sprint and Iteration
Customer Feedback	At the end of the project when the product is released	At the end of the project plus internally with each iteration	At the end of each release and iteration plus internally with each sprint
Retrospection	After each release	After each iteration or milestone	After each sprint
Quality Control	Detection & fixing primarily last phase of project	Early detection & fixing in each iteration and in last phase of project	Early detection & fixing in each sprint followed by stabilization