

Object-Oriented Programming

Course Number: CPSC-24500

Week: 1

Instructor: Eric Pogue

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. Recap: How is object-oriented programming different
14. Recap: Why did we choose to learn an object-oriented approach in developing software

We have a lot of solid topics for our first week. You will quickly notice that three words are underlined. You will see references to object-oriented concepts, patterns, and principles throughout the week and throughout the class.

Concepts: Practices, standards, and tools that effectively support object-oriented design and programming... I need to do enhance the BMI class to provide better Encapsulation and hide the height and weight properties (hours).

Patterns: Well established templates for forging relationships between classes... I would like to generate the random Animal generator using a Factory pattern so it is more supportable (days)

Principles: Proven industry guidelines... Using a Factory Pattern is a good start; however, I think we should focus more on making sure that we follow the Open Close Principle throughout our product (weeks+)

If you take a look through the topics you will hopefully see a few that are familiar; however, I suspect that many will be new. Don't be concerned, by the end of the week you will have some familiarity with all of them. You should also have resources that will let you go back and refresh your memory when you need to do that.

I am going to divide the topics up into three or four sections so that we can focus and keep our energy.

Learning Objectives – Session 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 concepts with examples
5. Distinguish between a class and an object

Section 1 will be focused on where object-oriented design and programming (OOP) fits into the Software Development Lifecycle, what platforms and tools we will be using as we explore OOP, and then get us started with classes and objects.

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 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.

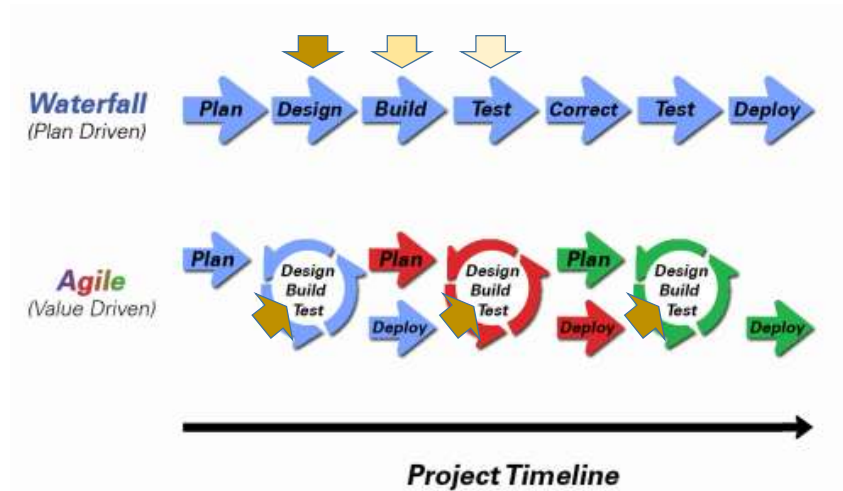
- Concepts – powerful features that prove indispensable to modern software development, brought to us automatically by object-oriented programming.
- Patterns – tried-and-true templates for forging relationships between classes
- Principles – guidelines that help you determine what classes are needed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

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. You should be able to click on the [link] tag (possibly while holding the shift key down) in order to open the reference in a browser. Please let me know if this is not working for you.

Walking through the slide, you will see that words and terminology will be important as we discuss and learn new concepts. During the course I am sure you will notice that at times I will struggle with the attribute/property 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.

For the purposes of this class, “attributes” and “properties” will be used interchangeably to describe variables belonging to a class or object. Also “procedures” and “functions” will be used interchangeably.

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. Can you start to see my biases?

Development Methodologies (SDLCs) are a future Bonus Topic. There are several optional slides at the end of this deck. Let me know if you would like to have a more formal overview of the topic as part of this class. I have a passion in this area.

Object oriented-programming concepts/practices evolve and reprioritize depending on the development methodology.








For example, 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. (opinion) 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.

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%

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. Let me know if you have a desire to do some Python work... or work in another OOP language. If so, we can likely work something out.

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 very sharp knife... use it carefully.

Object-Oriented Concepts 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:

$$\text{BMI} = \text{weight [kg]} / (\text{height [m]} * \text{height [m]})$$

Procedural BMI (body mass index):

Data:

- Height
- Weight

Procedures (or functions):

- CalcBMI

Object-Oriented BMI:

Class BMI

- Attributes
 - Height
 - Weight
- Methods
 - CalcBMI

Well let's get started on a real-world example that will allow us to see how OOP is used, and to learn some of the basic concepts like classes and objects. I've picked a body mass index (BMI) calculator as our first example. If retrospect, I wish I would have used something with shapes or animals. However, I assure you that there will be no shortage of shape and animal examples in the coming weeks.

Can you see how data and procedures/functions map directly to class attributes and methods? Even though it is a directly mapping, it is still good to get our terminology correct.

Example: Procedural vs. Object Oriented Programming

Procedural BMI (C):

```
// BMI Calculator (Procedural 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);
    }
}
```

Now let's look at some real code examples. You will quickly notice that I like to use real code examples over philosophical discussion topics. I will try to make the more important coding examples available to you on my GitHub account. Let me know if you see something and would like a copy.

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)...

We have a pretty simple formula...

When I coded this the first time I used 'int' instead of 'float' for all my variables. That didn't work very well.

If you look carefully you will see that we still have a logic problem in both of our implementations. Don't worry we are going to get back to fix that using some of our OOP concepts.

For now let's focus on looking through our first OOP Java implantation. We have our first class. Very nice, we will see many more of those.

Let's go to the next slide to take a closer look at our class.

Distinguish Between a Class and an Object

Procedural BMI (C):

```
// BMI Calculator (Procedural 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.

Now we need to be able to use our class to do something. For that we need to create an instance of our class.



Distinguish Between a Class and an Object

Procedural BMI (C):

```
// BMI Calculator (Procedural 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”.

We use or myBMI object to set some properties and then we have it calculate BMI by calling “my.BMI.CalcBMI()”

... and of course when we compile and run it, we get... the wrong answer. But it compiled and ran successfully. That’s something, right?

I know you can’t wait to see how this BMI story ends, so that will be our starting point for Section 2. How to fix BMI using object-oriented concepts/techniques.

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. Recap: How is object-oriented programming different
14. Recap: Why did we choose to learn an object-oriented approach in developing software

Welcome back for Section 2. In Section 1 we covered some important topics. We made our way through defining a BMI class that was pretty but didn't work as expected. Now we are going to learn some more OOP concepts and use those concepts to “fix” our BMI implementation. Well, to be fair the BMI implementation worked... as long as we used metric units.

Learning Objectives – Session 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

For Section 2 we are going to cover some OOP concepts, use those concepts to extend our BMI example, and then talk a little about UML.

The “six” (Three plus) Object-Oriented Concepts

Object-oriented concepts:

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

Plus... Composition & Aggregation

Helpful Interview Hint: 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 examples can be very effective... and it is always best to use animals in you OOP interview examples.

Now we just need to make sure that we are able to effectively utilize these concepts after we get the job. Let’s start by walking through an example.

Encapsulation & Information Hiding

Encapsulation: Wrapping properties and methods into a class and minimizing the scope of those properties and methods.

Information Hiding: Minimize visibility/scope of data, attributes, functions, and methods.

Inheritance & Abstraction

Inheritance: When one class acquires the properties and methods of another class it is called Inheritance.

Abstraction: 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.

```
// Inheritance
class Shape {
}

class Circle extends Shape {
}

class Rectangle extends Shape {
}
```

The first of many “Shape” examples. We will get to a Abstraction example in a few minutes.

Polymorphism

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.

```
// Polymorphism
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());
}
```

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.

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.

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

Procedural BMI (C):

```
// BMI Calculator (Procedural 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);
    }
}
```

Let's go back to where we left off in our BMI example. We had both Procedural (C) code and Object-Oriented Java code. Both compiled. Both ran. Neither produce the expected result because...

The function / method only worked if we used metric units where we were using English units. Specifically, our BMI formula assumes metric inputs of kg & m while our interactions with the BMI procedures and methods are using English units if 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... definitely one with animals.

Revising Procedural (C) BMI Implementation

Procedural BMI (C):

```
// BMI Calculator (Procedural 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:

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

float heightinches = 0;
float weightinlbs = 0;

float heightinm = 0;
float weightinkg = 0;

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

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

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

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

First let's spend a couple minutes revising our procedural C implementation the "old fashioned" way.

Since our current implementation actually works for metric units, let's not lose that. We will rename our variables to reflect that they are metric only units (heightinm and weightinkg). Note that since our variables were global, this change alone will break anyone else who was reusing our code (ouch!)

We will also rename our C function to reflect its Metric requirement.

Revising Procedural (C) BMI Implementation

Procedural BMI (C):

```
// BMI Calculator (Procedural 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:

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

float heightinches = 0;
float weightinlbs = 0;

float heightinm = 0;
float weightinkg = 0;

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

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

int main() {
    heightinches = (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)\ConsoleApplication1.exe
BMI: 25.678856

Now we will add in English variables and functions. Finally, we will call the new English function... and voilà! It works as expected. Hmmmm, 25.6? I thought it would be lower than that.

Not bad. However, when we do these examples, always imagine thousands or tens-of-thousands of lines of code. And recognize that as the size of the code grows and the number of developers grows that the complexity grows exponentially.

Now think about the opportunity to reuse this type of code. If this was a complex ten thousand line implementation, would you want your livelihood dependent on this code and it's owner? I would not want to be on call the Sunday night that he or she decided to make a last minute change and update the global variable name... and, of course, it would be my code that actually broke because my implementation would be dependent on the variable name remaining the same. This is one reason that the (very bad) practice of "copy-paste" code sharing has become so prevalent.

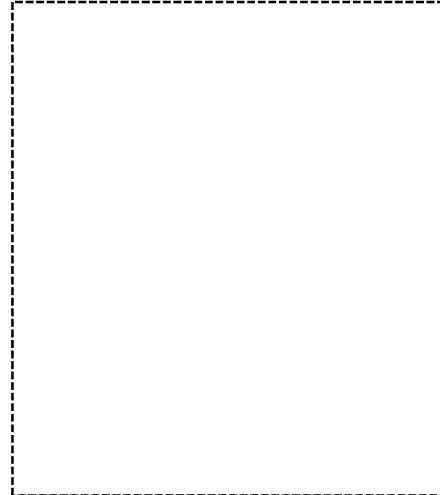
This is one of the many reasons that OOP has remained so prevalent over the years. It is not necessarily easier to write OOP code the first time (vs. procedural development); however, the opportunity for well written OOP code to be reused, extended, maintained, and efficiently tested has made it indispensable in modern software development.

Revising BMI Implementations

Procedural BMI (C) - revised:

```
/* *****  
// BMI Calculator (Procedural C)  
// BMI = weight over height squared  
  
float heightinches = 0;  
float weightinlbs = 0;  
  
float heightinm = 0;  
float weightinkg = 0;  
  
float CalcBMIMetric(void) {  
    return weightinkg / (heightinm * heightinm);  
}  
  
float CalcBMIEnglish(void) {  
    heightinm = heightinches * 0.025;  
    weightinkg = weightinlbs * 0.45;  
    return CalcBMIMetric();  
}  
  
int main() {  
    heightinches = (6.0 * 12.0) + 1; // 6'1"  
    weightinlbs = 190.0;           // 190 lbs  
  
    float BMI = CalcBMIEnglish();  
    printf("BMI: %f\n", BMI);  
    return 0;  
}
```

Object-Oriented BMI (revised):



Repeat from last slide: This is one of the many reasons that OOP has remained so prevalent over the years. It is not necessarily easier to write OOP code the first time (vs. procedural development); however, the opportunity for well written OOP code to be reused, extended, maintained, and efficiently tested has made it indispensable in modern software development.

Now let's revise our Java OOP BMI implementation.

Encapsulation

Object -Oriented BMI:

```
/**
 * 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:

```
/**
 * 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);
    }
}
```

First, let's Encapsulate our code.

#1 Rule for Encapsulation: Minimize class property and method scope and visibility: (1)none, (2)local, (3)method parameters, (4)private attribute, (5)protected attribute, and (6)public

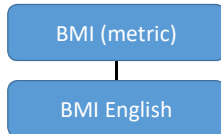
In this case we can't eliminate our height and width properties, and we can't make them local to our method. However, we can make them parameters to our CalcBMI method. That's a nice simplification. It also makes it very unlikely that future changes will impact users of our BMI class.

Consider: How would you add protective code around setting height to 0 in the initial code (height > 2ft & <9ft)? ... How about in our second encapsulated example? Once again consider reuse, testing, and additional modification if this were thousands of lines off code.

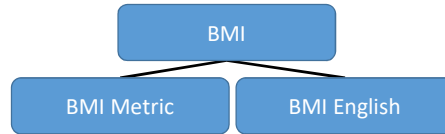
This type of Encapsulation & Information Hiding is a feature of nearly all modern development languages... not just object-oriented languages.

Inheritance... Options to Implement English Units

Option #1:



Option #2:

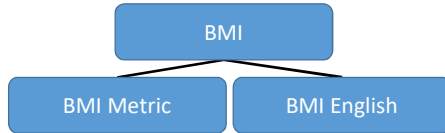


Now let's use Inheritance to implement English unit support in BMI. There are really two Inheritance class hierarchy options in this case. Option #1 is very practical and safe... and very unlikely to break anyone's code who is currently using our class. Option #2 is more pure and elegant.

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. Profession developer rule #1: Don't break what is working.

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



```
//*****  
// 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);  
    }  
}
```

Let's look at how we enhanced our class hierarchy. We created a new Abstract base class call BMI. Yes, the fact that we did this likely broke others that were using our class. We will need to call them and apologize. Note that implementing Option #1 on the proceeding slide should have avoided this. Notice the Abstract CalcBMI method in BMI. We have not learned the difference between Abstract and Virtual yet. Virtual means a Method can be overridden where Abstract means it must be overridden.

We also implemented two classes that extend BMI. Our original class was renamed to "BMIMetric" and we added a new class "BMIEnglish"... seem elegant and simple.

Now we create a new BMI that is actually a BMIEnglish. If that doesn't mess with you mind just a little, think about it some more. Note that we could have made the line:

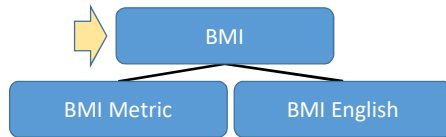
"BMI myBMI = new BMIEnglish();" read

"BMIEnglish myBMIEnglish = new BMIEnglish();"

And have gotten the same results. I actually think the second line is cleaner and simpler. It avoids Polymorphism... which really isn't need here.

If you understand why these two line produce the same results in this example? If so you are well on your way to understanding Abstraction and Polymorphism. Or you might want to come back to this after our Polymorphism example and see if it makes more sense.

Superclass and Subclass



➡ Superclass

```
/*=====
// 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);
    }
}

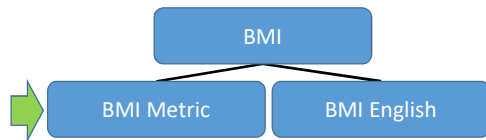
class BMIEnglish extends BMI {
    public float CalcBMI(float height, float weight) {
        // Convert to meters and kg.
        height = height * (float)0.025;
        weight = weight * (float)0.45;
        return weight / (height * height);
    }
}

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);
    }
}
```

Let's backtrack and cover a couple required (and valuable) items.
We now have an example of a Superclass in BMI

Superclass and Subclass



➡ 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);
    }
}

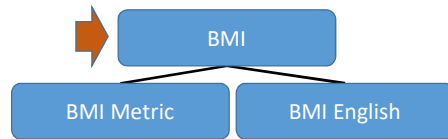
class BMIEnglish extends BMI {
    public float CalcBMI(float height, float weight) {
        // Convert to meters and kg.
        height = height * (float)0.025;
        weight = weight * (float)0.45;
        return weight / (height * height);
    }
}

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);
    }
}
```

And we have two examples of Subclasses. That was easy.

Abstraction



➡ Abstraction

```
/*=====
// 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);
    }
}
```

And we have an example of Abstraction.

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 abstract class 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 interface 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.

```
graph TD; Animal --> Dog; Animal --> Cat; Cat --> Bigcat;
```

```

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("Hissssss!");
    }
}

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

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(Bark!);
            if (randAnimalIndex == 0) someAnimal.Grrrr... Hissssss!
            else if (randAnimalIndex == 1) soHissssss!
            else if (randAnimalIndex == 2) soHissssss!
            else if (randAnimalIndex == 3) soHissssss!
            else if (randAnimalIndex == 4) soHissssss!
            else if (randAnimalIndex == 5) soHissssss!
            else if (randAnimalIndex == 6) soHissssss!
            else if (randAnimalIndex == 7) soHissssss!
            else if (randAnimalIndex == 8) soHissssss!
            else if (randAnimalIndex == 9) soHissssss!
            else if (randAnimalIndex == 10) soHissssss!
            else if (randAnimalIndex == 11) soHissssss!
            else if (randAnimalIndex == 12) soHissssss!
            else if (randAnimalIndex == 13) soHissssss!
            else if (randAnimalIndex == 14) soHissssss!
            else if (randAnimalIndex == 15) soHissssss!
            else if (randAnimalIndex == 16) soHissssss!
            else if (randAnimalIndex == 17) soHissssss!
            else if (randAnimalIndex == 18) soHissssss!
            else if (randAnimalIndex == 19) soHissssss!
            else if (randAnimalIndex == 20) soHissssss!
            else if (randAnimalIndex == 21) soHissssss!
            else if (randAnimalIndex == 22) soHissssss!
            else if (randAnimalIndex == 23) soHissssss!
            else if (randAnimalIndex == 24) soHissssss!
            else if (randAnimalIndex == 25) soHissssss!
            else if (randAnimalIndex == 26) soHissssss!
            else if (randAnimalIndex == 27) soHissssss!
            else if (randAnimalIndex == 28) soHissssss!
            else if (randAnimalIndex == 29) soHissssss!
            else if (randAnimalIndex == 30) soHissssss!
            else if (randAnimalIndex == 31) soHissssss!
            else if (randAnimalIndex == 32) soHissssss!
            else if (randAnimalIndex == 33) soHissssss!
            else if (randAnimalIndex == 34) soHissssss!
            else if (randAnimalIndex == 35) soHissssss!
            else if (randAnimalIndex == 36) soHissssss!
            else if (randAnimalIndex == 37) soHissssss!
            else if (randAnimalIndex == 38) soHissssss!
            else if (randAnimalIndex == 39) soHissssss!
            else if (randAnimalIndex == 40) soHissssss!
            else if (randAnimalIndex == 41) soHissssss!
            else if (randAnimalIndex == 42) soHissssss!
            else if (randAnimalIndex == 43) soHissssss!
            else if (randAnimalIndex == 44) soHissssss!
            else if (randAnimalIndex == 45) soHissssss!
            else if (randAnimalIndex == 46) soHissssss!
            else if (randAnimalIndex == 47) soHissssss!
            else if (randAnimalIndex == 48) soHissssss!
            else if (randAnimalIndex == 49) soHissssss!
        }
    }
}

```

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.

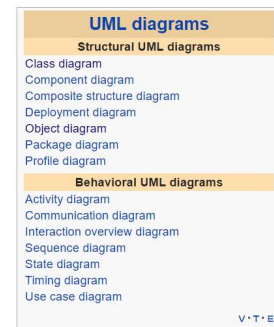
The easy way to tell if something is composition or aggregation is to ask if the owner is responsible for creating and destroying the thing that is owned. If so, it's a composition relationship.

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)



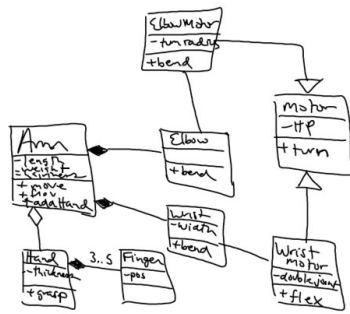
UML can describe many different things and become quite complex. There is some debate, particularly in the Agile development methodology circles, as to the benefits of this documentation outweigh the cost. Most developers agree that at least a basic understanding of UML diagrams aids greatly in the impromptu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

```

// *****
// 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. Recap: How is object-oriented programming different
14. Recap: Why did we choose to learn an object-oriented approach in developing software

There ends Section 2. We have covered a lot including object-oriented concepts, encapsulation, inheritance, polymorphism, superclass, subclass, inheritance, abstraction, aggregation, composition, UML, and likely a few other topics as well.

It's a lot. Be sure to take a break and come back to finish out this week's topics with patterns, principles, and a recap of the week.

Be forewarned, even though patterns and principles are “one-liners”, there's a lot there. It will likely take as much time on thought to get through Section 3 as it did for us to get through Sections 1 and 2.

Learning Objectives - Session 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. Recap: How is object-oriented programming is different
14. Recap: Why did we choose to learn an object-oriented approach in developing software

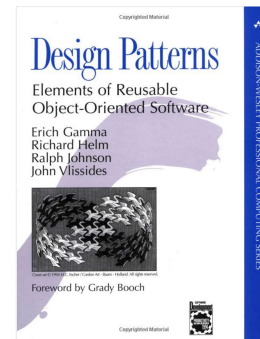
We made it to patterns, principles, and a recap of the week. These are big topics and will likely take us about as long to get through as sections 1 and 2.

Object-Oriented Design Patterns

Definition: A software design pattern is a commonly repeated approach to constructing software. These approaches are commonly repeated because they consistently produce quality results.

Common Patterns Include:

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



“Design Patterns: Elements of Reusable Object-Oriented Software” has been influential in defining software engineering patterns and is regarded as an important source for object-oriented design theory and practice. The authors are often referred to the “Gang of Four (GoF)”

Interview tip: The answer to any question that references design patterns should include “Gang of Four”, “Reusable Object-Oriented Software”, and “Model-View-Controller”. Now let’s learn what those are.

Why Use Patterns?

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

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 are pretty easy to understand and come in very handy. A application Logfile class or Configuration file manager are common examples of singletons.

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 or making sure there is a single point of control for a limited resource like a file or printer.

A singleton can be written in Java 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.

Factory Design Pattern

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

```
*****
// 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");
```

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.

Delegation Design Pattern

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

[\[link\]](#)

“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();
    }
}
```

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

[[Left up to interested reader to complete... or ask for the code.]]

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.

```
////////////////////////////////////  
// 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());  
}
```

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 that there will likely be many different Views that access the same data including:

student view
faculty view
administrator view,
Web student view,
mobile student view, etc.

Evolution of UI and Data segregation

- Document-View (View was responsible for View-Controller functionality)
- Model-View-Controller
- Model-View-Viewmodel

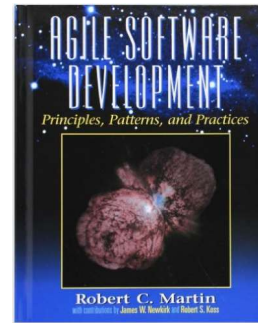
Learn this Pattern!

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.



Our procedural (C) implementation of BMI would be a great example of Immobility... maybe a good example of all three of these.

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

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();
    }
}
```

Our example of BMI is C was a great example of NOT implementing this principle. We had to modify the core functionality in order to extend it. It take a long time to know if a class exhibits this principle. You only know after a class has been used by multiple other classes (preferably “owned” by other developers), the class has need to be extended (preferably multiple times), and the dependent classes have not had to change (and the dependent developers have not complained).

Dependency Inversion Principle

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

```
//*****
// 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();
        }
    }
}
```

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.

Interface Segregation Principle

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

```
// 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()
}
```

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.

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's 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: Squire should 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;
    }
}
```

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. The patterns and principles should remain (largely) consistent across languages and platforms.

Recap: Object-Oriented Programming

- Rich is practice and theory with well accepted concepts, patterns, and principles
- Consistent support for powerful concepts including features that support:
 - ❑ Encapsulation... and Information Hiding
 - ❑ Inheritance... and Abstraction
 - ❑ Polymorphism
- Strong support for delivering extensible, maintainable, supportable, and scalable solutions
- Long history of industry success in managing complexity and delivering solutions that minimize rigidity, fragility, and immobility.
- Encompasses both design and programming activities
- Provides benefit across various development methodologies

Recap: Why Choose an Object-Oriented Approach

#1 – This is the best way we know to consistently deliver high quality software products

#2 – It is the approach demanded by the industry where we hope to be employed

#3 – It's an enjoyable way to build, enhance, and support software solutions

You may have different reasons why you think it is important to learn object-oriented design and programming. That's okay.

End of Session

Course Number: CPSC-24500

Week: 1

Instructor: Eric Pogue

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