Object Oriented Programming

What is OO?

The first object oriented programming languages started to appear in the 1960s, but at the time were not very popular. It is only recently (in the last 15 years or so) that object oriented programming has become popular.

Object-oriented programming (OOP) is a style of programming that focuses on using objects to design and build applications. Think of an object as a model of the concepts, processes, or things in the real world that are meaningful to your application. For example, in a project management application, you would have a status object, a cost object, and a client object among others. These objects would work together (and with many other objects) to provide the functionality that you want your project management application to have.

The four principles of object-oriented programming are encapsulation, abstraction, inheritance, and polymorphism.

OO brings us

- Ease of design
- Increased productivity
- OOP provides a clear modular structure for programs.
- It is good for defining abstract data types.
- Implementation details are hidden from other modules and other modules have a clearly defined interface.
- objects, methods, instantiation, message passing, inheritance, encapsulation, polymorphism and abstraction are some important properties provided by these particular languages
- It implements real-life manufacturing principles
- In OOP, the programmer not only defines data types but also deals with operations applied for data structures.

OOP breaks down your programming solution into smaller, more manageable chunks. These chunks are rather like Lego bricks: they have a clear interface (represented by messages), and you compose a program by snapping these bricks (or objects) together.

The object's internal state and implementation are not visible to the external world, so this makes it much easier to understand what you're dealing with. You only need to understand messages.

Objects can also be reused across applications; the reuse of software also lowers the cost of development. More effort is put into the object-oriented analysis and design, which lowers the overall cost of development.

OO makes software easier to maintain: Since the design is modular, part of the system can be updated in case of issues, without a need to make large-scale changes

Better Productivity

As OOP techniques enforce rules on a programmer that, in the long run, help them get more work done; finished programs can work better, have more features and are easier to read and maintain. OO programmers take new and existing software objects and "stitch" them together to make new programs. Because object libraries contain many useful functions, software developers don't have to reinvent the wheel as often; more of their time goes into making the new program.

Object-oriented programming takes the view that what we really care about are the objects we want to manipulate rather than the logic required to manipulate them. Examples of objects range from human beings (described by name, address, and so forth) to buildings and floors (whose properties can be described and managed) down to the widgets on a computer desktop (such as buttons and scroll bars).

The first step in OOP is to identify all the objects the programmer wants to manipulate and how they relate to each other, an exercise often known as data modeling. Once an object has been identified, it is generalised as a class of objects (think of Plato's concept of the "ideal" chair that stands for all chairs) which defines the kind of data it contains and any logic sequences that can manipulate it. Each distinct logic sequence is known as a method. Objects communicate with well-defined interfaces called messages.

Object-oriented programming is used to develop many applications—simple and complex applications, business applications and games, mobile and desktop applications. Developers choose to program in the object-oriented paradigm because the proper use of objects makes it easier to build, maintain, and upgrade an application. Also, developing an application with objects that have been tested increases the reliability of the application.

Objects

In programming terms, an object is a self-contained component that contains properties and methods needed to make a certain type of data useful. An object's properties are what it knows and its methods are what it can do. For example, a project management application could have a status object, a cost object, and a client object, among others. One property of the status object would be the current status of the project. The status object would have a method that could update that status. The client object's properties would include all of the important details about the client and its methods would be able to change them. The cost object would have methods necessary to calculate the project's cost based on hours worked, hourly rate, materials cost, and fees.

In addition to providing the functionality of the application, methods ensure that an object's data is used appropriately by running checks for the specific type of data being used. They also allow for the actual implementation of tasks to be hidden and for particular operations to be

standardised across different types of objects. You will learn more about these important capabilities when we talk about Encapsulation.

Objects are the fundamental building blocks of applications from an object-oriented perspective. You will use many objects of many different types in any application you develop. Each different type of object comes from a specific class of that type.

Classes, instances, and instantiation

A class is a blueprint or template or set of instructions to build a specific type of object. Every object is built from a class. Each class should be designed and programmed to accomplish one, and only one, thing (this is called 'single responsibility principle'). Because each class is designed to have only a single responsibility, many classes are used to build an entire application.

An instance is a specific object built from a specific class. It is assigned to a reference variable that is used to access all of the instance's properties and methods. When you make a new instance the process is called instantiation and is typically done using the 'new' keyword.

Think about classes, instances, and instantiation like baking a cake. A class is like a recipe for chocolate cake. The recipe itself is not a cake. You can't eat the recipe (or at least wouldn't want to). If you correctly do what the recipe tells you to do (instantiate it) then you have an edible cake. That edible cake is an instance of the chocolate cake class.

You can bake as many cakes as you would like using the same chocolate cake recipe. Likewise, you can instantiate as many instances of a class as you would like. Pretend you are baking three cakes for three friends who all have the same birthday but are different ages. You will need some way to keep track of which cake is for which friend so you can put on the correct number of candles. A simple solution is to write each friend's name on the cake. Reference variables work in a similar fashion. A reference variable provides a unique name for each instance of a class. In order to work with a particular instance, you use the reference variable it is assigned to.

Writing a class

All objects come from a class: A class is a blueprint or template or set of instructions to build a specific type of object. Every object is built from a class. Going back to our chocolate cake; A class is like a recipe. You can't eat the recipe but you can eat the instance of the recipe that is a result of following the recipe instructions. Look at how a chocolate cake class might appear in code:

```
public class ChocolateCake {
    public var numberOfCandles:int;
    public function ChocolateCake(){
    }

    public function putCandlesOnCake():void{
        trace("Putting " + numberOfCandles + " candles on the cake.");
    }
}
```

Class declaration:

In the example above,

public class ChocolateCake {…}

is the class declaration statement. The class keyword tells the compiler that a class is being defined. It is preceded by an attribute keyword (public in this example) and followed by the class name (ChocolateCake).

Class attributes are similar to the access control attributes used for variables and functions. When part of the class declaration, the attributes modify the entire class.

Table 1. Class attributes

Attributes	Definition

dynamic	Allow properties to be added to instances at runtime.
final	Must not be extended by another class.
internal (default)	Visible to references inside the current package.
public	Visible to references everywhere.

Class names follow similar naming conventions as variables. The main difference to note is that class names always start with a capital letter. The name of the class should be descriptive. ChocolateCake,ProjectStatus, ProjectClient, CustomButton, ShoppingCartItem, and ProgramPiece are all examples of good class names.

The name of the source file must be the same as the class (with the relevant file extension appended). For example, the file that defines this ChocolateCake class must be named ChocolateCake.jar. The class declaration statement will also include the keyword ‘extends’ followed by a class name or ‘implements’ followed by an interface name if inheritance or an interface is being used.

Class body with a constructor method

All of the code that is contained within the curly brackets that follow the class declaration is considered the class body. Remember, objects are self-contained components that contain properties and methods needed to make a certain type of data useful. The class body contains those properties and methods. Here is the class body for the ChocolateCake example:

```
public var numberOfCandles:int;

public function ChocolateCake(){

}


public function putCandlesOnCake():void{

    trace("Putting " + numberOfCandles + " candles on the cake.");

}
```

In code, properties are variables declared within a class. The ChocolateCake class only has one property: numberOfCandles . Methods are functions defined within a class. ChocolateCake has two methods: putCandlesOnCake() and the constructor method ChocolateCake().

Classes can have a constructor method. The constructor method is a method that is automatically called upon instantiation. When you use the ‘new’ keyword to create a new instance of a class, you are calling the constructor method of the class. Any code included in the constructor method is executed every time the class is instantiated.

In this example, the class name is ChocolateCake so the name of the constructor method is also ChocolateCake.

Instance versus static properties and methods

Objects instantiated from the same class will have the same properties and methods. Continuing with the chocolate cake example, every instance of the ChocolateCake class will have a numberOfCandles property and a putCandlesOnCake() method. However, the value stored within the numberOfCandles property can vary for each instance. If every instance of the class had the same value for the property, every cake would have the same number of candles every time you baked it. That would not be very useful over a long period of time. Properties that vary for each instance are called instance properties.

Instance methods provide functionality that is useful for each instance of the class. Because putCandlesOnCake() is an instance method, each instance of the ChocolateCake class can have candles on it. If you don’t want candles on a specific instance, you wouldn’t call the putCandlesOnCake() method for that instance.

Instance properties and methods are accessed by using dot syntax with the reference variable name and the name of the property or method. If you instantiated the ChocolateCake class and assigned it to a reference variable named myCake, you would use myCake.numberOfCandles and myCake.putCandlesOnCake() to access the instance property and method of the class.

You will notice the ‘this’ keyword in the code below. The ‘this’ keyword is used to reference the instance itself. It can be used to reference anything defined at the class level. In the example below, it is being used to reference an instance property.

Static properties and methods belong to the class rather than an instance of the class. Static properties are properties that describe the class, not just a specific instance. In a revised version of the ChocolateCake class below, the property ‘flavour’ is a static property. This property will tell you what flavour the recipe is without having to go through the process of instantiating.

```
package cakes {
    public class ChocolateCake {

    public static var flavour:String = "Chocolate";
    public var numberOfCandles:int;

    public function ChocolateCake(candles:int){
        this.numberOfCandles = candles;
    }

        public function putCandlesOnCake():void{
            trace("Putting " + numberOfCandles + " candles on the cake.");
        }
    }
}
```

Similarly, static methods encapsulate functionality that does not affect individual instances of the class. It shouldn't make sense to apply the functionality in a static method to an instance of the class. Leaving the ChocolateCake class for a moment, consider the Date class. It has a static method named parse(), which takes a string and converts it to a number. The method is static because it does not affect an individual instance of the class. Instead, the parse() method takes a string that represents a date value, parses the string, and returns a number in a format compatible with the internal representation of a Date object. This method is not an instance method, because it does not make sense to apply the method to an instance of the Date class.

Contrast the static parse() method with one of the instance methods of the Date class, such as getMonth().The getMonth(); method is an instance method, because it operates directly on the value of an instance by retrieving a specific component, the month, of a Date instance.

Static properties and methods are declared by using the 'static' keyword. Because they belong to the class, they are accessed using the class name rather than a reference variable. The static property for the chocolate cake class would be accessed by ChocolateCake.flavour. The static parse() method of the Date class mentioned above would be accessed by Date.parse().

SRP, reusability, and encapsulation

Some of the principles of object-oriented programming are principles of class design. Using well-designed classes helps to increase the stability and maintainability of your application. Well-designed objects only have a single responsibility and are reusable, flexible, and encapsulated.

The Single Responsibility Principle (SRP) states that a class should never have more than one responsibility. If it does have multiple responsibilities, changes to one responsibility may unintentionally break the other responsibilities in the class. This can lead to a domino effect in your application, causing it to perform or break in unpredictable ways.

When designing objects, keep reusability and flexibility in mind. You want to write classes that can be used over and over again within your application, and possibly in future ones. Ask yourself if the object you are designing can be used in the next version of the application with few, if any, changes. Can it work if the application is reconfigured? Can it work in another application? You will be able to answer 'yes' to at least one of these questions if you are building a reusable, flexible object. When you begin using the principle of customisation through inheritance, your objects can achieve an even higher degree of reusability.

In the real world, objects frequently hide their information and how they work. They have a public interface that allows you to successfully use the object without knowing anything about how it works. You don't need to know the internal details in order to use the object as long as you know how to use its public interface. Think about a light switch. In order to turn on the lights, you simply toggle the switch. You don't need to be an electrician or understand anything about electricity to work the electrical system in a room. An automobile is another good example. As long as you know how to use the accelerator and brake pedals, steering wheel and ignition, you can drive. You don't have to understand how the engine, or transmission, or electrical system works to drive a car. Nor do you need to know the specific properties of the car such as engine size, horsepower, or torque. In object-oriented programming, well-designed objects encapsulate their data and functionality from other objects. We will discuss encapsulation a bit more later.

Below are three versions of a rectangle class designed to draw a rectangle. Each version of the class brings it closer to achieving the goals of SRP, reusability, flexibility, and encapsulation.

Rectangle class version one

Look at the class MyRectangle_v1 in the code below. It has a single responsibility - To draw a rectangle. Every instance of this class will draw a rectangle when its draw() method is called. Because literal data is used in the draw method for colour, size, and location, the rectangle drawn will always be the same size, the same colour, and at the same location. Although the class does have a single responsibility, the fact that every instance will draw an identical rectangle reduces the reusability and flexibility of the code.

```
package {
    import flash.display.Sprite;
    public class MyRectangle_v1 extends Sprite {
        public function MyRectangle_v1() {
        }

        public function draw():void{
            graphics.lineStyle(1);
            graphics.beginFill(0xff0000);
            graphics.drawRect(100, 100, 75, 25);
            graphics.endFill();
        }
    }
}
```

Rectangle class version two

One way to make the class more reusable and flexible is by adding parameters to the

draw() method for the rectangle's size, colour, and location. By adding these parameters, the rectangle's size, colour, and location can be varied each time the draw() method is called.

```
package {
    import flash.display.Sprite;
    public class MyRectangle_v2 extends Sprite {
        public function MyRectangle_v2() {
        }

        public function draw(outlineWeight:Number, colour:uint, xLocation:int, yLocation:int, width:int, height:int):void{
            graphics.lineStyle(outlineWeight);
            graphics.beginFill(colour);
            graphics.drawRect(xLocation, yLocation, width, height);
            graphics.endFill();
        }
    }
```

```
}
```

Rectangle class version three

This rectangle class is reusable, flexible, and has a single responsibility - three of the goals of object-oriented programming. In the final version, look at the first step toward encapsulating the important properties of the rectangle. To encapsulate the properties from other classes you use the keyword ‘private’ during variable declaration. This is only the first step towards encapsulation: there is more you have to do to fully encapsulate the object. The concept and process of encapsulation will be covered shortly.

Private properties are still accessible from within the class itself. Below, values passed in as arguments to the constructor method are assigned to the private properties. Those properties are used as arguments for the method calls within the draw() method.

```
package {
    import flash.display.Sprite;

    public class MyRectangle_v3 extends Sprite {
        private var _outlineWeight:Number;
        private var _colour:uint;
        private var _xLocation:int;
        private var _yLocation:int;
        private var _rectangleWidth:int;
        private var _rectangleHeight:int;

        public function MyRectangle_v3(outlineWeight:Number, colour:uint, xLocation:int, yLocation:int, rectangleWidth:int, rectangleHeight:int)
{
            _outlineWeight = outlineWeight;
            _colour = colour;
            _xLocation = xLocation;
            _yLocation = yLocation;
            _rectangleWidth = rectangleWidth;
            _rectangleHeight = rectangleHeight;
        }

        public function draw():void{
            graphics.lineStyle(_outlineWeight);
            graphics.beginFill(_colour);
            graphics.drawRect(_xLocation, _yLocation, _rectangleWidth, _rectangleHeight);
            graphics.endFill();
        }
    }
}
```

Encapsulation

Put simply, encapsulation is about hiding complexity. In the real world, objects frequently hide their information and how they work. You don't need to know the internal details in order to use an object.

When you create an object in an object-oriented language, you can hide the complexity of the internal workings of the object. As a developer, there are two main reasons why you would choose to hide complexity.

The first reason is to provide a simplified and understandable way to use your object without the need to understand the complexity inside. As mentioned above, a driver doesn't need to know how an internal combustion engine works. It is sufficient to know how to start the car, how to engage the transmission if you want to move, how to provide fuel, how to stop the car, and how to turn off the engine. You know to use the key, the gearstick, the clutch, the accelerator and the brake pedal to accomplish each of these operations. These basic operations form an interface for the car. Think of an interface as the collection of things you can do to the car without knowing how each of those things works.

Hiding the complexity of the car from the user allows anyone, not just a mechanic, to drive a car. In the same way, hiding the complex functionality of your object from the user allows anyone to use it and to find ways to reuse it in the future regardless of their knowledge of the internal workings. This concept of keeping implementation details hidden from the rest of the system is key to object-oriented design.

Take a look at the CombustionEngine class below. Notice that it has only two public methods: start() and stop()

Those public methods can be called from outside of the object. All of the other functions are private, meaning that they are not publicly visible to the rest of the application and cannot be called from outside of the object.

```
package engine {
    public class CombustionEngine {
        public function CombustionEngine() {}
        private function engageChoke():void {}
        private function disengageChoke():void {}
        private function engageElectricSystem():void {}
        private function powerSolenoid():void {}
        private function provideFuel():void {}
        private function provideSpark():void {}

        public function start():void {
            engageChoke();
            engageElectricSystem();
            powerSolenoid();
            provideFuel();
            provideSpark();
            disengageChoke();
        }
      public function stop():void {}
    }
}
```

You would use this class as follows:

```
var carEngine:CombustionEngine = new CombustionEngine();
```

carEngine.start();

carEngine.stop();

The second reason for hiding complexity is to manage change. Today most of us who drive use a vehicle with a petrol-powered internal combustion engine. However, there a petrol-electric hybrids, pure electric motors, and a variety of internal combustion engines that use alternative fuels. Each of those engine types has a different internal mechanism yet we are able to drive each of them because that complexity has been hidden. This means that, even though the mechanism which propels the car changes, the system itself functions the same way from the user's perspective.

Imagine a relatively complex object that parses an audio file and yet allows you only to play, seek or stop the playback. Over time, the author of this object could change the internal algorithm of how the object works completely - new optimisation for speed could be added, memory handling could be improved, additional file formats could be supported, and the like. However, since the rest of the source code in your application uses only the ‘play’, ‘seek’ and ‘stop’ methods, this object can change very significantly internally while the remainder of your application can stay exactly the same. This technique of providing encapsulated code is critical in team environments.

Take a look at the refactored CombustionEngine class below. Over time, technology has improved. Instead of having a manual choke and carburetor, it now uses fuel injection. The internal system has therefore changed dramatically.

Notice that it still only has two public methods:

start()

and

stop()

The rest of your system will remain the same even though the internals of your engine changed. Imagine that you have a relatively complex application that uses the engine from hundreds of places. Not having to update each of those places when you change engines could save many days of work.

```
package engine {
    public class CombustionEngine {
        public function CombustionEngine() {}
        private function engageElectricSystem():void {}
        private function powerComputer():void {}
        private function powerSolenoid():void {}
        private function provideFuel():void {}
        private function provideSpark():void {}

        public function start():void {
            engageElectricSystem();
            powerComputer();
            powerSolenoid();
            provideFuel();
            provideSpark();
        }

        public function stop():void {}
    }
}
```

You would use this class as follows:

```
var carEngine:CombustionEngine = new CombustionEngine();
```

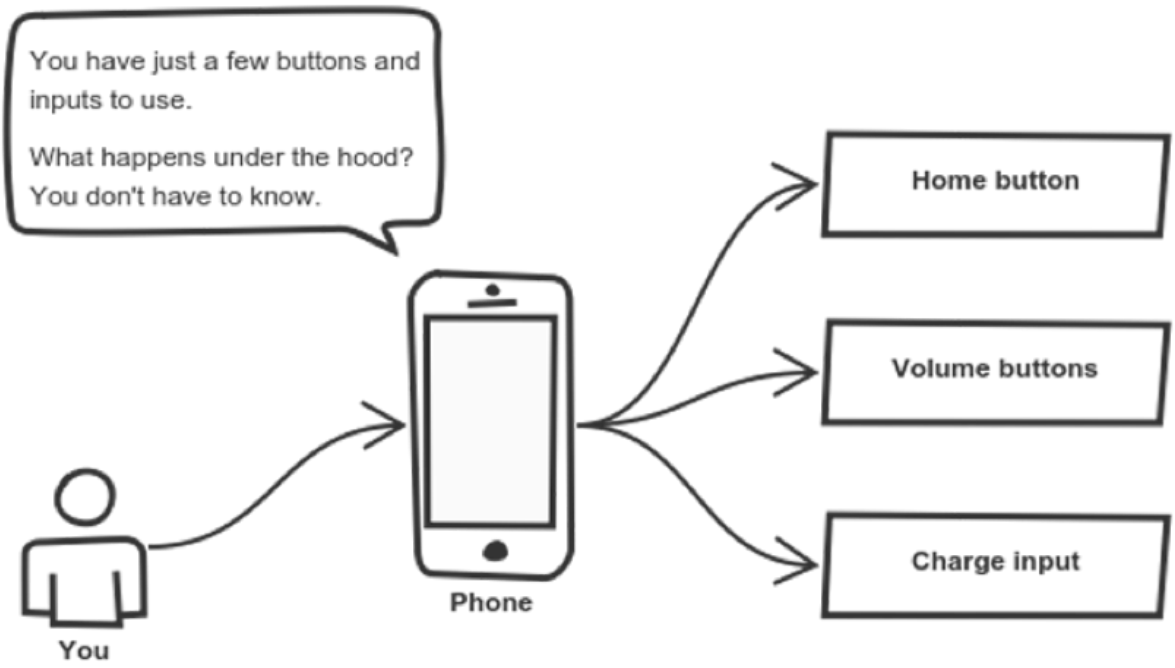
```
carEngine.start();

carEngine.stop();
```

Abstraction

Abstraction can be thought of as a natural extension of encapsulation. In object-oriented design, programs are often extremely large. And separate objects communicate with each other a lot. So maintaining a large codebase like this for years—with changes along the way—is difficult. Abstraction is a concept aiming to ease this problem.

Applying abstraction means that each object should **only** expose a high-level mechanism for using it. This mechanism should hide internal implementation details. It should only reveal operations relevant for the other objects. Think—a coffee machine. It does a lot of stuff and makes quirky noises under the hood. But all you have to do is put in coffee and press a button. Preferably, this mechanism should be easy to use and should rarely change over time. Think of it as a small set of public methods which any other class can call without "knowing" how they work.

Another real-life example of abstraction?



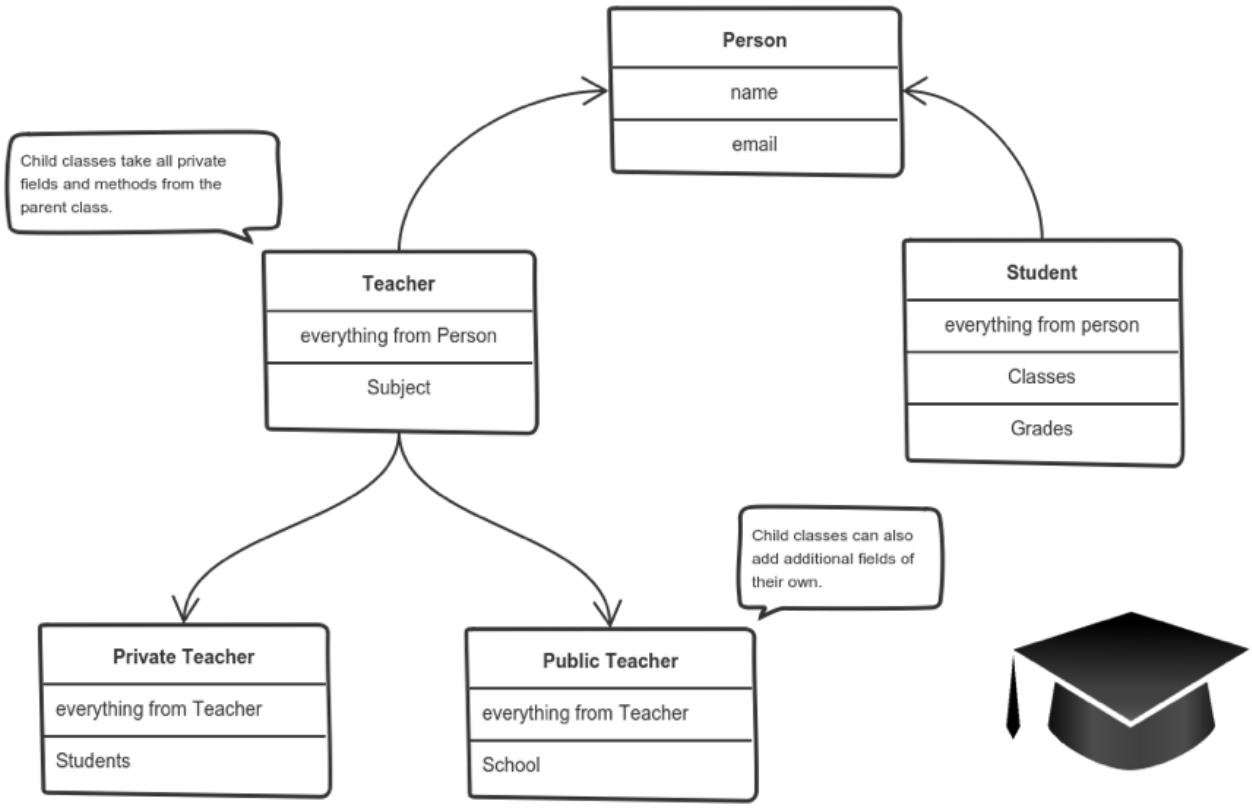
Think about how you use your phone. You interact with your phone by using only a few buttons. What's going on under the hood? You don't have to know—implementation details are hidden. You only need to know a short set of actions. Implementation changes—for example, a software update—rarely affect the abstraction you use.

Inheritance

OK, we saw how encapsulation and abstraction can help us develop and maintain a big codebase. But do you know what is another common problem in OOP design?

Objects are often very similar. They share common logic. But they're not **entirely** the same. So how do we reuse the common logic and extract the unique logic into a separate class? One way to achieve this is inheritance. It means that you create a (child) class by deriving from another (parent) class. This way, we form a hierarchy. The child class reuses all fields and methods of the parent class (common part) and can implement its own (unique part).

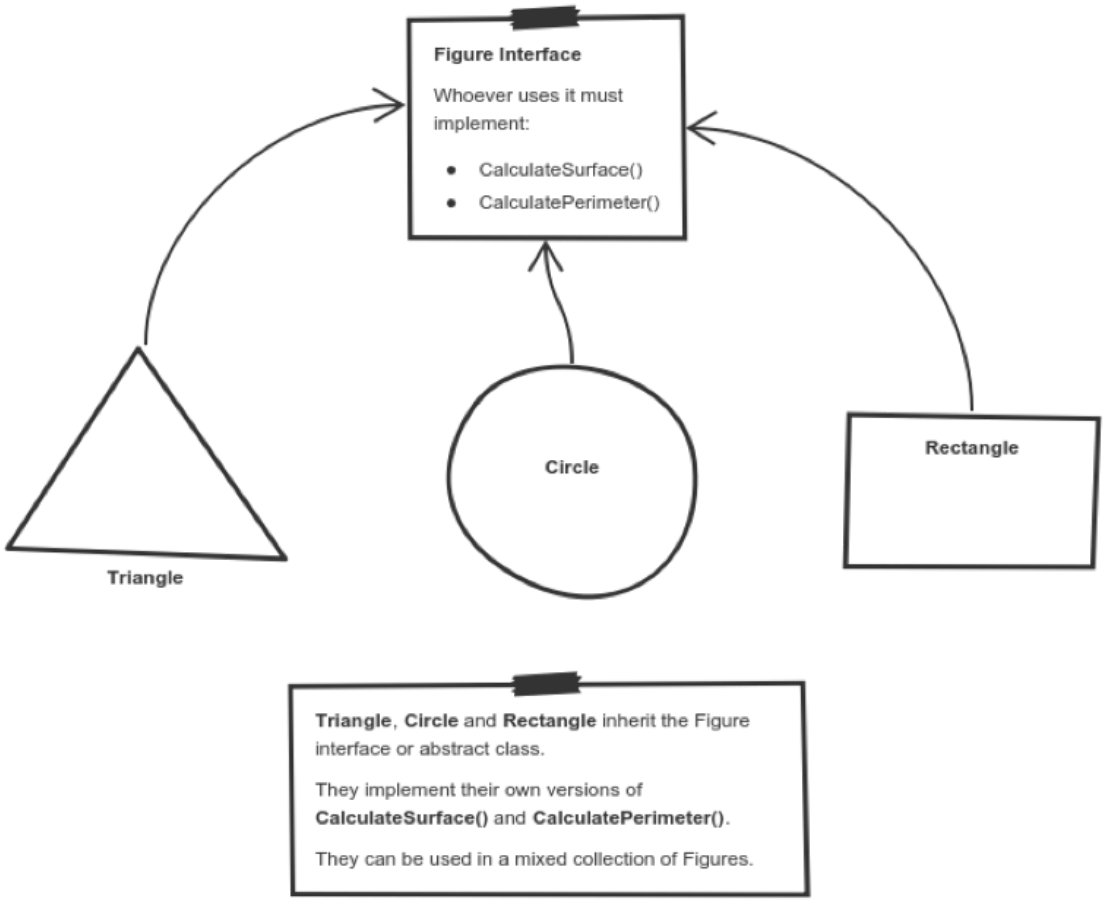
For example:



A private teacher is a type of teacher. And any teacher is a type of Person. If our program needs to manage public and private teachers, but also other types of people like students, we can implement this class hierarchy. This way, each class adds only what is necessary for it while reusing common logic with the parent classes.

Polymorphism

We're down to the most complex word! Polymorphism means "many shapes" in Greek. So we already know the power of inheritance and happily use it. But there comes this problem: Say we have a parent class and a few child classes which inherit from it. Sometimes we want to use a collection—for example a list—which contains a mix of all these classes. Or we have a method implemented for the parent class—but we'd like to use it for the children, too... This can be solved by using polymorphism. Simply put, polymorphism gives a way to use a class exactly like its parent so there's no confusion with mixing types. But each child class keeps its own methods as they are. This typically happens by defining a (parent) interface to be reused. It outlines a bunch of common methods. Then, each child class implements its own version of these methods. Any time a collection (such as a list) or a method expects an instance of the parent (where common methods are outlined), the language takes care of evaluating the right implementation of the common method—regardless of which child is passed.



Take a look at a sketch of geometric figures implementation. They reuse a common interface for calculating surface area and perimeter: Triangle, Circle, and Rectangle now can be used in the same collection. Having these three figures inheriting the parent Figure Interface lets you create a list of mixed triangles, circles, and rectangles. And treat them like the same type of object. Then, if this list attempts to calculate the surface for an element, the correct method is found and executed. If the element is a triangle, triangle's CalculateSurface() is called. If it's a circle—then circle's CalculateSurface() is called. And so on.

If you have a function which operates with a figure by using its parameter, you don't have to define it three times—once for a triangle, a circle, and a rectangle. You can define it once and accept a Figure as an argument. Whether you pass a triangle, circle or a rectangle—as long as they implement CalculateParamter(), their type doesn't matter.

Differences between Go and a 'true' OO language

Structs Instead of Classes

Go does not provide classes but it does provide structs. Methods can be added on structs. This provides the behaviour of bundling the data and methods that operate on the data together akin to a class.

Composition Instead of Inheritance

Now read

https://golang.org/doc/code.html

Down to the section titled '**Your first library'**

Now read

https://gobyexample.com/

For next session please complete the exercises up to and including ‘Arrays’. Email myself (Alan) and Maryl if you have any problems.

Polymorphism in Go:

https://www.ardanlabs.com/blog/2013/07/object-oriented-programming-in-go.html