Classes and Objects

What is an Object

Object **O**riented **P**rogramming is a programming style that is intended to make thinking about programming closer to thinking about the real world.

In programming, **objects** are independent units, and each has its own **identity**, just as objects in the real world do.

> An apple is an object; so is a mug. Each has its unique **identity**. It's possible to have two mugs that look identical, but they are still separate, unique objects.

Objects

An object might contain other objects but they're still different objects.

Objects also have **characteristics** that are used to describe them. For example, a car can be red or blue, a mug can be full or empty, and so on. These characteristics are also called **attributes**. An attribute describes the current **state** of an object.
Objects can have multiple attributes (the mug can be **empty**, **red** and **large**).

> An object's **state** is independent of its type; a cup might be full of water, another might be empty.

Objects

In the real world, each object **behaves** in its own way. The car **moves**, the phone **rings**, and so on. The same applies to objects - behavior is specific to the object's type.

> So, the following three dimensions describe any object in object oriented programming: **identity**, **attributes**, **behavior**.

Objects

In programming, an object is **self-contained**, with its own **identity**. It is separate from other objects.
Each object has its own **attributes**, which describe its current state. Each exhibits its own **behavior**, which demonstrates what they can do.

In computing, objects aren't always representative of physical items.
For example, a programming object can represent a date, a time, a bank account. A bank account is not tangible; you can't see it or touch it, but it's still a well-defined object - it has its own **identity**, **attributes**, and **behavior**.

> Tap **Continue** to dive right into Object Oriented Programming (**OOP**) with C++!

What is a Class

Objects are created using **classes**, which are actually the focal point of OOP.

The class **describes** what the object will be, but is separate from the object itself.
In other words, a class can be described as an object's **blueprint**, description, or definition.
You can use the same class as a blueprint for creating multiple different objects. For example, in preparation to creating a new building, the architect creates a blueprint, which is used as a basis for actually building the structure. That same blueprint can be used to create multiple buildings.

Programming works in the same fashion. We first define a class, which becomes the blueprint for creating objects.

Each class has a **name**, and describes **attributes** and **behavior**.

In programming, the term **type** is used to refer to a class name: We're creating an object of a particular **type**.

> Attributes are also referred to as **properties** or **data**.

Methods

Method is another term for a class' behavior. A method is basically a **function** that belongs to a class.

> Methods are similar to functions - they are blocks of code that are called, and they can also perform actions and return values.

A Class Example

For example, if we are creating a banking program, we can give our class the following characteristics:
name: BankAccount
attributes: accountNumber, balance, dateOpened
behavior: open(), close(), deposit()

The class specifies that each object should have the defined attributes and behavior. However, it doesn't specify what the actual data is; it only provides a **definition**.

Once we've written the class, we can move on to create objects that are based on that class. Each object is called an instance of a class. The process of creating objects is called **instantiation**.

> Each object has its own identity, data, and behavior.

Declaring a Class

Begin your class definition with the keyword **class**. Follow the keyword with the class name and the class body, enclosed in a set of curly braces.
The following code declares a class called **BankAccount**:

```
class BankAccount {

};
```

A class definition must be followed by a **semicolon**.

Declaring a Class

Define all **attributes** and **behavior** (or members) in the body of the class, within curly braces.
You can also define an **access specifier** for members of the class.
A member that has been defined using the **public** keyword can be accessed from outside the class, as long as it's anywhere within the scope of the class object.

You can also designate a class' members as **private** or **protected**. This will be discussed in greater detail later in the course.

Creating a Class

Let's create a class with one public method, and have it print out "Hi".

```
class BankAccount {
  public:
    void sayHi() {
      cout << "Hi" << endl;
    }
};
```

The next step is to instantiate an object of our **BankAccount** class, in the same way we define variables of a type, the difference being that our object's type will be **BankAccount**.

```
int main()
{
  BankAccount test;
  test.sayHi();
}
```

Try It Yourself

Our object named **test** has all the members of the class defined.
Notice the **dot separator (.)** that is used to access and call the method of the object.

We must declare a class **before** using it, as we do with functions.

Abstraction

Data **abstraction** is the concept of providing only essential information to the outside world. It's a process of representing essential features **without including implementation details**.

A good real-world example is a *book*: When you hear the term book, you don't know the exact specifics, i.e.: the page count, the color, the size, but you understand **the idea of a book** - the abstraction of the book.

> The concept of **abstraction** is that we focus on essential qualities, rather than the specific characteristics of one particular example.

Abstraction

Abstraction means, that we can have an idea or a concept that is completely separate from any specific instance.
It is one of the fundamental building blocks of object oriented programming.

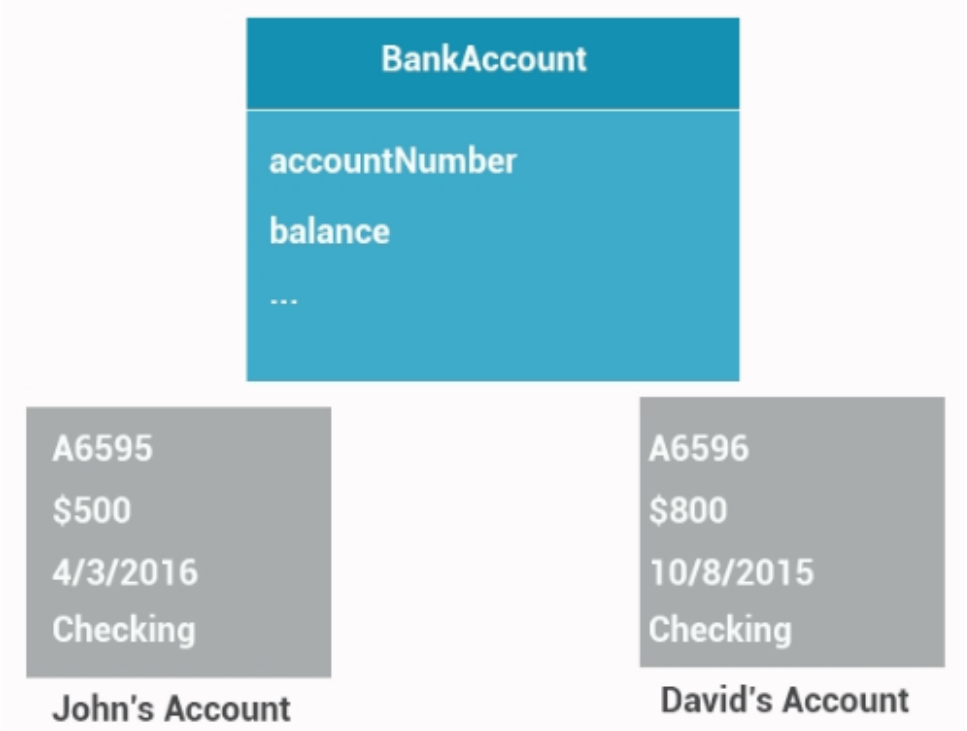
For example, when you use cout, you're actually using the cout object of the class **ostream**. This streams data to result in standard output.

```
cout << "Hello!" << endl;
```

> In this example, there is no need to understand how **cout** will display the text on the user's screen. The only thing you need to know to be able to use it is the public interface.

Abstraction

Abstraction allows us to write a single bank account class, and then create different objects based on the class, for individual bank accounts, rather than creating a separate class for each bank account.



Abstraction acts as a foundation for the other object orientation fundamentals, such as **inheritance** and **polymorphism**. These will be discussed later in the course.

Encapsulation

Part of the meaning of the word encapsulation is the idea of "surrounding" an entity, not just to keep what's inside together, but also to **protect** it.
In object orientation, encapsulation means more than simply combining attributes and behavior together within a class; it also means restricting access to the inner workings of that class.

The key principle here is that an object only reveals what the other application components require to effectively run the application. All else is kept out of view.

This is called **data hiding**.

Encapsulation

For example, if we take our **BankAccount** class, we do not want some other part of our program to reach in and change the **balance** of any object, without going through the **deposit()** or **withdraw()** behaviors.
We should **hide** that attribute, control access to it, so it is accessible only by the object itself.
This way, the **balance** cannot be directly changed from outside of the object and is accessible only using its methods.
This is also known as "**black boxing**", which refers to closing the inner working zones of the object, except of the pieces that we want to make public.
This allows us to change attributes and implementation of methods without altering the overall program. For example, we can come back later and change the data type of the **balance** attribute.

In summary the benefits of encapsulation are:
- **Control** the way data is accessed or modified.
- Code is more **flexible** and easy to change with new requirements.
- **Change** one part of code without affecting other part of code.

Access Specifiers

Access specifiers are used to set access levels to particular members of the class.
The three levels of access specifiers are **public**, **protected**, and **private**.

A **public** member is accessible from outside the class, and anywhere within the scope of the class object.

For example:

```
#include <iostream>
#include <string>
using namespace std;

class myClass {
  public:
    string name;
};

int main() {
  myClass myObj;
  myObj.name = "SoloLearn";
```

```
  cout << myObj.name;
  return 0;
}

//Outputs "SoloLearn"
```

Try It Yourself

The **name** attribute is **public**; it can be accessed and modified from outside the code.

> Access modifiers only need to be declared once; multiple members can follow a single access modifier.
> Notice the **colon** (:) that follows the **public** keyword.

Private

A **private** member cannot be accessed, or even viewed, from outside the class; it can be accessed only from within the class.
A **public** member function may be used to access the **private** members. For example:

```
#include <iostream>
#include <string>
using namespace std;

class myClass {
  public:
    void setName(string x) {
      name = x;
    }
  private:
    string name;
};

int main() {
  myClass myObj;
  myObj.setName("John");

  return 0;
}
```

Try It Yourself

The **name** attribute is **private** and not accessible from the outside.
The public **setName()** method is used to set the **name** attribute.

> If no access specifier is defined, all members of a class are set to **private** by default.

Access Specifiers

We can add another public method in order to get the value of the attribute.

```
class myClass {
  public:
    void setName(string x) {
      name = x;
    }
```

```
  string getName() {
    return name;
  }
 private:
  string name;
};
```

The **getName()** method returns the value of the private **name** attribute.

Access Specifiers

Putting it all together:

```
#include <iostream>
#include <string>
using namespace std;

class myClass {
  public:
    void setName(string x) {
      name = x;
    }
    string getName() {
      return name;
    }
  private:
    string name;
};

int main() {
  myClass myObj;
  myObj.setName("John");
  cout << myObj.getName();

  return 0;
}

//Outputs "John"
```

Try It Yourself

We used encapsulation to hide the **name** attribute from the outside code. Then we provided access to it using public methods. Our class data can be read and modified only through those methods.

This allows for changes to the implementation of the methods and attributes, without affecting the outside code.

Constructors

Class **constructors** are special member functions of a class. They are executed whenever new objects are created within that class.

The constructor's name is identical to that of the class. It has no return type, not even void.

For example:

```
class myClass {
  public:
    myClass() {
      cout <<"Hey";
    }
    void setName(string x) {
      name = x;
    }
    string getName() {
      return name;
    }
  private:
    string name;
};

int main() {
  myClass myObj;

  return 0;
}

//Outputs "Hey"
```

Try It Yourself

Now, upon the creation of an object of type **myClass**, the constructor is automatically called.

Constructors

Constructors can be very useful for setting initial values for certain member variables.
A default constructor has no parameters. However, when needed, parameters can be added to a constructor. This makes it possible to assign an initial value to an object when it's created, as shown in the following example:

```
class myClass {
  public:
    myClass(string nm) {
      setName(nm);
    }
    void setName(string x) {
      name = x;
    }
    string getName() {
      return name;
    }
  private:
    string name;
};
```

We defined a constructor, that takes one parameter and assigns it to the **name** attribute using the **setName()** method.

Constructors

When creating an object, you now need to pass the constructor's parameter, as you would when calling a function:

```
class myClass {
  public:
    myClass(string nm) {
      setName(nm);
    }
    void setName(string x) {
      name = x;
    }
    string getName() {
      return name;
    }
  private:
    string name;
};

int main() {
  myClass ob1("David");
  myClass ob2("Amy");
  cout << ob1.getName();
}
//Outputs "David"
```

Try It Yourself

We've defined **two** objects, and used the constructor to pass the initial value for the **name** attribute for each object.

It's possible to have multiple constructors that take different numbers of parameters.

End.