SOFTWARE GUILD

All About Objects

.NET Cohort

SOFTWARE GUILD

Lesson Goals

- Introduce the basic concepts of Object-Oriented Programming (OOP)
- Describe the relationship between instances of objects and classes

We Love Abstraction

Every programming language provides some level of abstraction.

What we mean by this is that we don’t want to think about programming problems in concrete computer terms, we want to think about them in terms of the problem we are trying to solve.

That’s a Lot of Fancy Words…

Would you rather read this?

```
         .486p
         .model  flat,STDCALL
include  win32.inc

extrn            MessageBoxA:PROC
extrn            ExitProcess:PROC

.data

HelloWorld db "Hello, world!",0
msgTitle db "Hello world program",0

.code
Start:
         push    MB_ICONQUESTION + MB_APPLMODAL + MB_OK
         push    offset msgTitle
         push    offset HelloWorld
         push    0
         call    MessageBoxA

         push 0
         call ExitProcess
ends
end Start
```

Or this?

```
class Messager
{
    static void SayHello()
    {
        MessageBox.Show("Hello World");
    }
}
```

So What Are Objects?

The “object-oriented” approach is used by programmers to define elements in the problem space.

Objects are the nouns of our world, and we name and organize them by useful units. For example, a sales program may have Products, SalesOrders, SalesPersons, Customers, etc.

When you need to express new ideas, you simply add new types of objects. We’re only limited by our own creativity.

Objects Contain…

In C# (and other object-oriented languages) objects contain two concepts:

Data Members — data in memory that the object needs to keep track of. For example, a Customer object might have a Name, Address, and PhoneNumber. (These are also commonly called fields.)

Function Members — actions an object can perform. A BankAccount object might have Deposit, Withdraw, and Transfer actions. (These actions are commonly called functions, procedures or methods.)

Five Characteristics of Objects

Objects in C# have 5 major characteristics:

1. Everything is an object and every object can be uniquely identified from other objects in memory.
2. A program is a collection of objects that send messages to each other.
3. Any object can be made up of other objects.
4. Every object has a type.
5. All objects of a specific type can store the same data and receive the same messages.

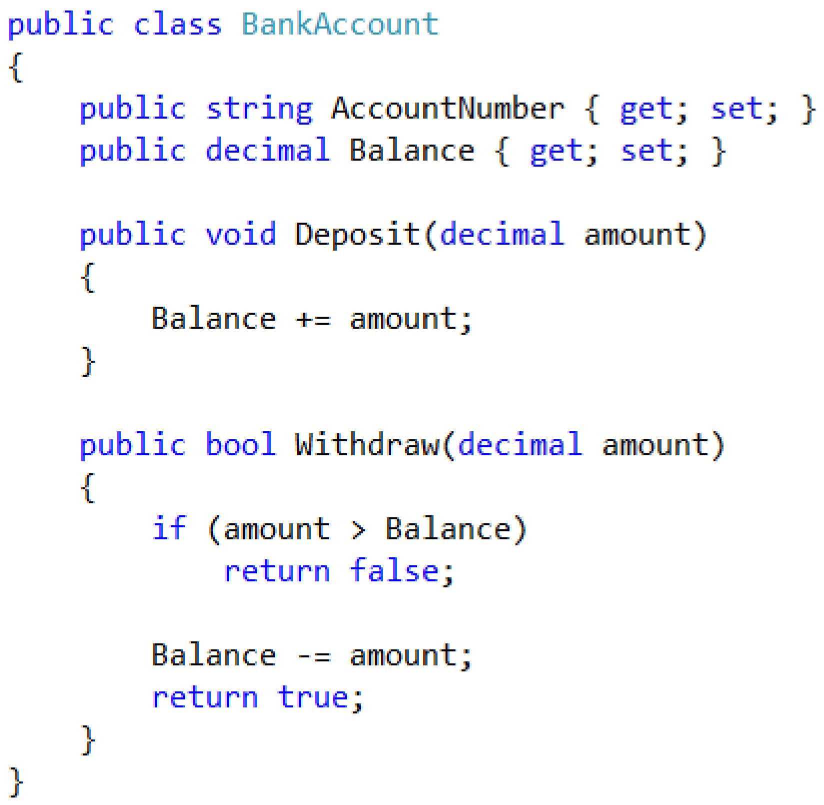
Everything is an object

Most people understand primitive variables, like int and string. These are objects too!

An object in C# is created from a class definition. The class is the blueprint that describes what data and behaviors an object will have when it is *instantiated* (created).

In the code to the right we declare a simple BankAccount class. Notice how it declares properties (AccountNumber and Balance) as well as function members (Deposit and Withdraw).

In C#, strings and decimals are objects too… everything is an object! So this object contains other objects!



```
public class BankAccount
{
    public string AccountNumber { get; set; }
    public decimal Balance { get; set; }

    public void Deposit(decimal amount)
    {
        Balance += amount;
    }

    public bool Withdraw(decimal amount)
    {
        if (amount > Balance)
            return false;

        Balance -= amount;
        return true;
    }
}
```

Every object can be uniquely identified in memory

Let's instantiate a couple bank accounts.

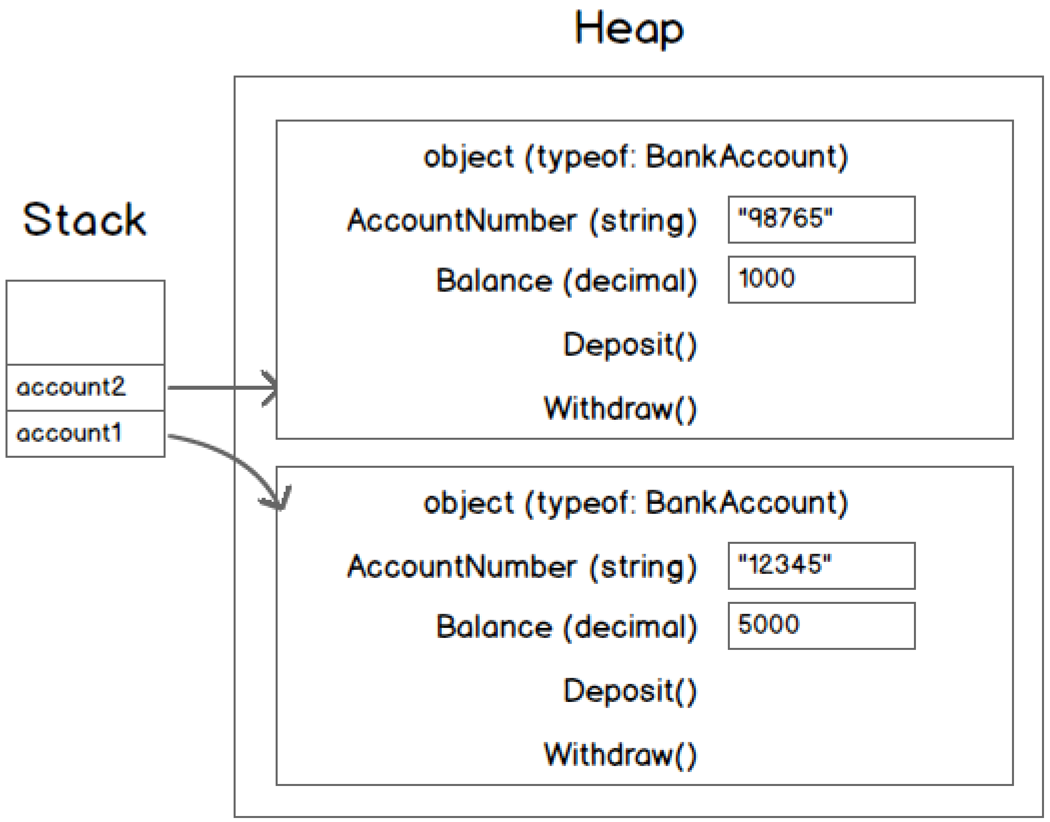
Up top, we see code that creates two new BankAccount objects. Below, we see our first glimpse of how our class objects are stored in memory.

Before using a non-primitive, non-static type, we must use the new keyword to instantiate the object.

We said previously that each object can be uniquely identified in memory and so our variables account1 and account2 each *point* to a unique object in memory.

Each object is referenced by the variable name, and <u>each object has its own copy of the data and methods</u>.

```
BankAccount account1 = new BankAccount();
account1.AccountNumber = "12345";
account1.Balance = 5000M;  // M means decimal

BankAccount account2 = new BankAccount();
account2.AccountNumber = "98765";
account2.Balance = 1000M;
```



A program is a collection of objects that send messages to each other

Here is a very simple program that flips a coin three times.

We created a class that defines a coin. It has a hidden data member (Random number generator) and only one method, called Flip(), that returns the string Heads or Tails, depending on the random number generated.

When a console application runs, a program object is created and its Main() method is called. This object creates a coin object and calls the Flip method.

In programmer speak, we might say:

“The program object’s main method instantiates a coin object and invokes its flip method three times.”

```
class Program
{
    static void Main()
    {
        Coin c = new Coin();

        Console.WriteLine("You flip the coin... it comes up {0}", c.Flip());
        Console.WriteLine("You flip the coin... it comes up {0}", c.Flip());
        Console.WriteLine("You flip the coin... it comes up {0}", c.Flip());
    }
}

public class Coin
{
    private Random rng = new Random();

    public string Flip()
    {
        int num = rng.Next(0, 2);

        if (num == 0)
            return "Tails";
        else
            return "Heads";
    }
}
```

Objects can be made up of other objects

We've already seen that an object like a BankAccount can contain objects such as strings and decimals.

The power of OO programming is that our objects can contain other objects that we have defined.

For example, look at the LibraryBook to the right and notice how it contains a list of Author objects.

So dividing data and methods into class definitions to create objects helps us organize our code and make those objects more closely model the real world.

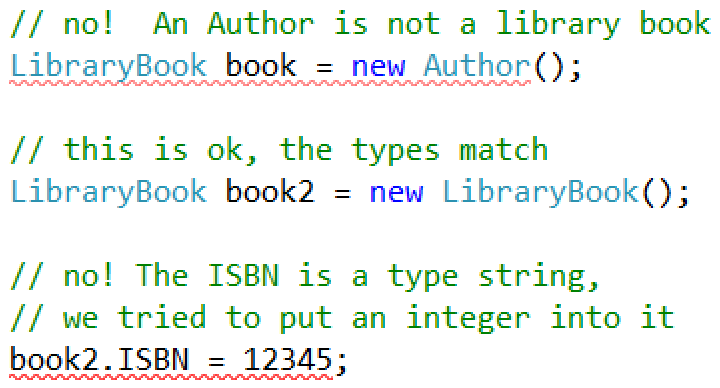
```
public class LibraryBook
{
    public string ISBN { get; set; }
    public string Title { get; set; }
    public List<Author> Authors { get; set; }
}

public class Author
{
    public int AuthorId { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public DateTime DateOfBirth { get; set; }
}
```

Every object has a type

C# is a statically-typed language, which means that the compiler requires you to declare the type of the object being stored in memory; after declaration, it will only allow objects of that type to be stored in that memory space.

In the code to the right, the C# compiler in Visual Studio will not allow us to assign an Author object to a LibraryBook variable.

It also won’t let us put an integer into a string property of a book object.



```
// no!  An Author is not a library book
LibraryBook book = new Author();

// this is ok, the types match
LibraryBook book2 = new LibraryBook();

// no! The ISBN is a type string,
// we tried to put an integer into it
book2.ISBN = 12345;
```

All objects of a specific type can store the same data and have the same messages

This comes into play when we talk about inheritance and interfaces later.

Simply, for now, inheritance means that a class can inherit (share) its class definition with another class.

The inheriting class, or child, gets everything that the parent class allows it to have.

In the code to the right, a Minivan object and a Car object are both Vehicles, and thus have a top speed.

We will cover this in more detail later.

```
public class Vehicle
{
    public int TopSpeed { get; set; }
}

public class Minivan : Vehicle
{
    public Minivan()
    {
        TopSpeed = 85;
    }
}

public class Car : Vehicle
{
    public Car()
    {
        TopSpeed = 105;
    }
}
```

Objects of a Type are Identical

For example: in our bank account program, each teller, customer, account, transaction, etc. is identical… except for their state during program execution.

Objects that are identical are grouped together into “classes of objects,” which is where the keyword class came from.

We can create variables of a type (these are objects or instances of objects). These variables have common members (data/functions). Every account has a balance, every teller can accept a deposit, etc.

Every instance has its own state. For example: each account has its own balance, each teller has an employee id. So, every object is a unique entity in your computer program.

Synonyms

Concept	What else developers call them
Class	Class definition, Type
Object	Instance of type
Class Member	Data, Functions
Data Member	Property, Field, Member
Function Member	Behavior, Method, Function, Subroutine, Procedure

An Object Provides Services

As beginners, the biggest struggle is trying to understand a program design. We recommend thinking about objects as “service providers.”

Your program provides services to the user. It will accomplish this by using the services of many objects.

Example: Contact List

We want our objects to have *high cohesion*. In other words, the members of an object should “fit together” well.

For example: in a small app that keeps track of your contacts, we may want an object that represents a contact, another that displays contact information to the user, and another that loads and saves contact information to a file or database.

The single responsibility principle means that objects should not mix “concerns.” For example: the object that displays information on the screen shouldn’t be concerned with how data is saved/loaded from a file. Just as every station in an assembly line has a unique task, each object should only be concerned with its own purpose and ask other objects to fulfill other needed tasks.

Three Kinds of Services

It’s helpful to break objects into three different responsibilities:

1. Data Transfer Objects (DTOs) — contain only data and validation of that data. They exist to shuttle information between objects.
2. Service/Manager/Operation Objects — contain methods to perform tasks (do calculations, call databases, etc.)
3. Workflow Objects — manage a process from start to finish. They orchestrate a process by calling the helper objects in the right order and passing DTOs as necessary.