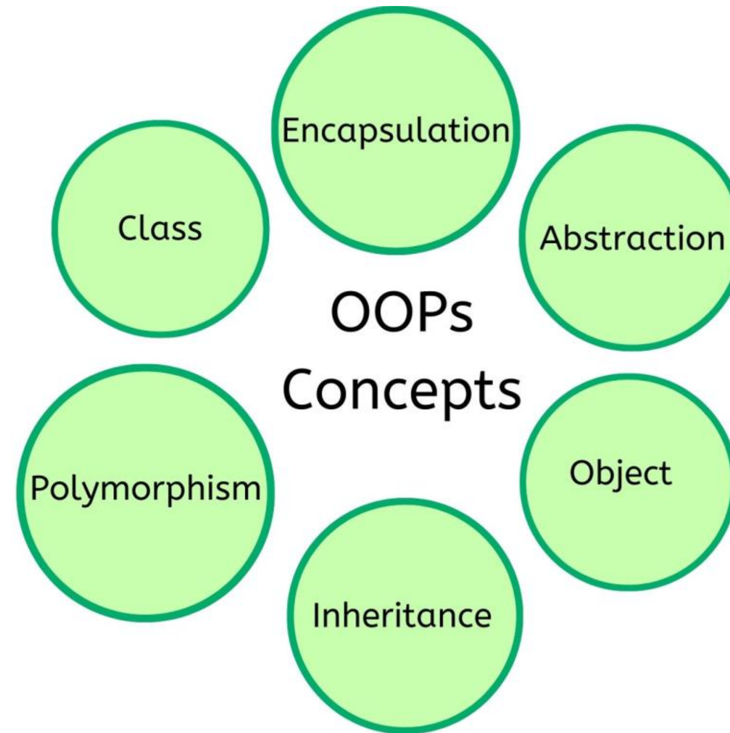
IT Department
ISRA University

Object Oriented Programming

Part I



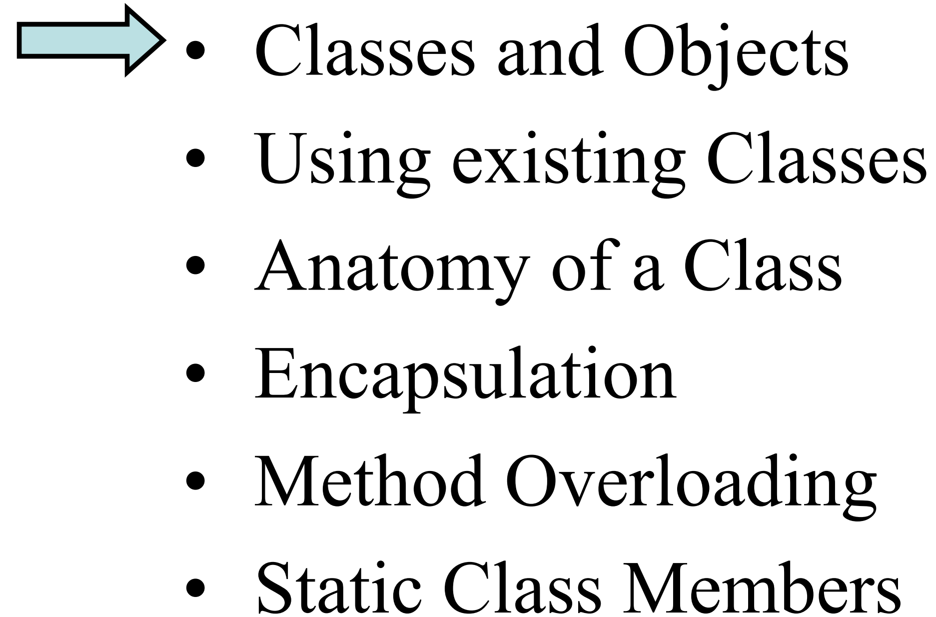
Section : 1
Instructor : Khalil Barhoum

Chapter 5 – Writing Classes

- We've been using predefined classes. Now we will learn to write our own classes to define objects

- Chapter 5 focuses on
  - class definitions and relationships
  - instance data
  - encapsulation and C# modifiers
  - constructors
  - method design and overloading

Outline



- Classes and Objects
- Using existing Classes
- Anatomy of a Class
- Encapsulation
- Method Overloading
- Static Class Members

Classes and Object

- The programs we've written in previous examples have used classes defined in the c# standard class library such as Console, Math and Random classes
- Now we will begin to design programs that rely on classes that we write ourselves
- Classes we define ourselves are called programmer-defined classes.
- The class that contains the `main` method is just the starting point of a program
- True object-oriented programming is based on defining classes that represent objects with well-defined characteristics and functionality

Classes and Objects

Classes

- **A class Specify the <u>common</u> structure(**data**) and behavior of a set of objects.**
- **A class <u>encapsulates</u>** attributes(variables) **and** operations(methods).
- **Each attribute has a** type**.**
- **Each operation has a** signature
- **A class is a blueprint from which individual objects are created.**

Objects

- **Object is a collection of <u>related</u> variables and methods**
- **are instances of classes,**
- **are created, modified, and destroyed during the execution of the system,**
- **have a state(variables + values=state)**

Concepts and Phenomena

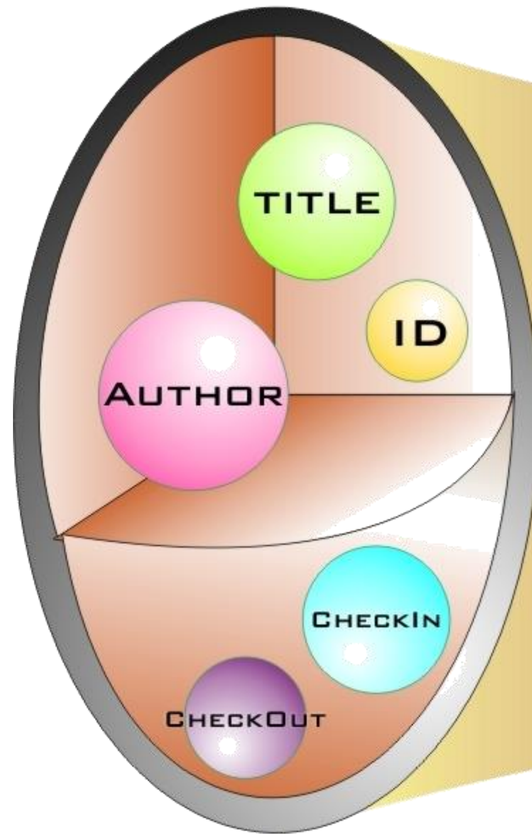
- A Phenomenon **is an object of the world as it is perceived, for example:**
  - **Classroom M-1-409**
  - **Professor Bob Morris**
  - **September 5**
- A Concept **is an abstraction describing a set of phenomena, for instance:**
  - **Classrooms**
  - **ISRA professors**
  - **Dates**
- A class **represents a concept.**
- **An instance represents a phenomenon**

Identifying Classes and Objects

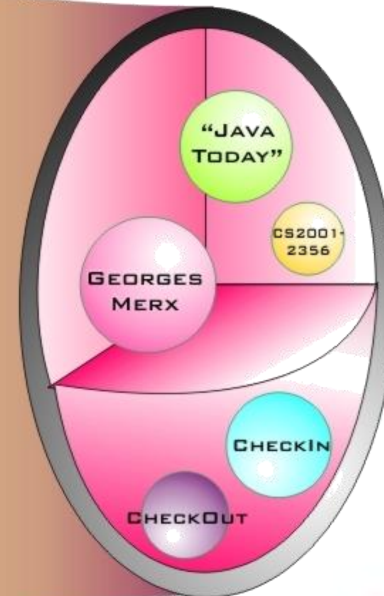
- Recall that object has *state* and *behavior*
- Consider a library book object
  - it's state can be defined as the book's title, author, ID.
  - it's primary behavior may be to checkin and checkout.
- We can represent a book in software by designing a class called `Book` that models this state and behavior
  - the class serves as the blueprint for a book object
- We can then instantiate as many book objects as we need for any particular program

Visualization

Class **Book**



Object **myBook**



Classes

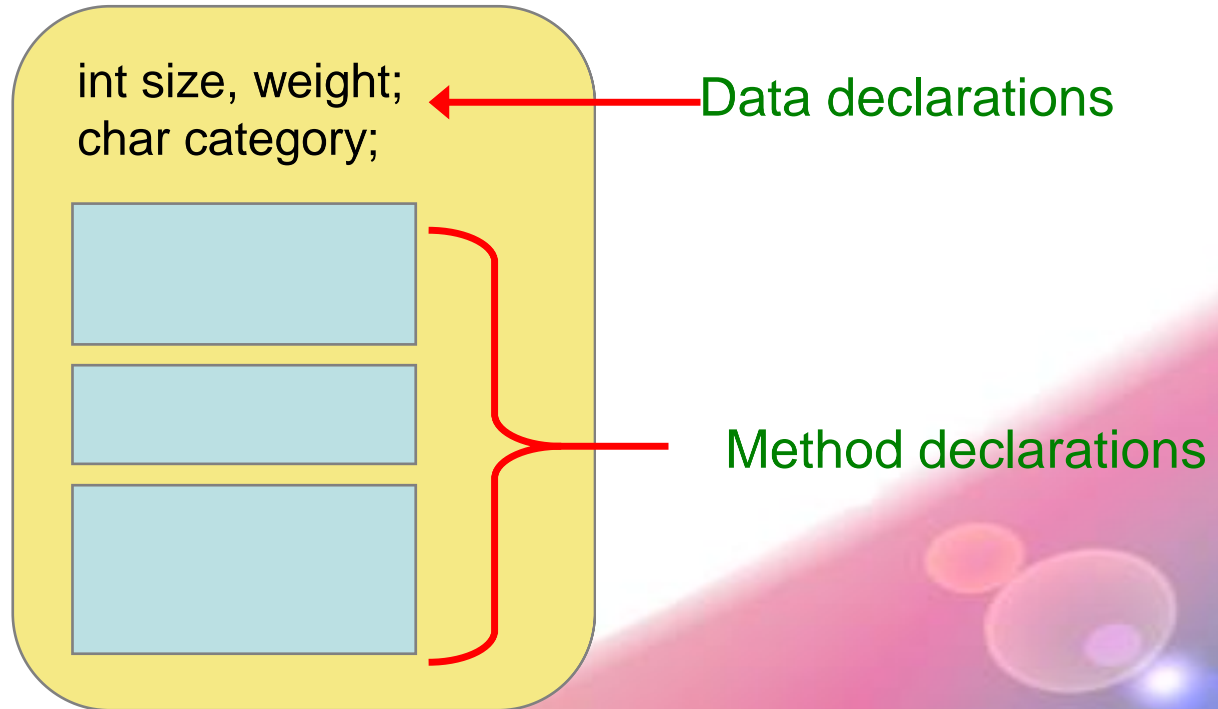
- **A class can contain data declarations and method declarations**

The values of the data define the state of an object created from the class

The functionality of the methods define the behaviors of the object

For our `Book` class, we might declare a `String` that represents the book's title

One of the methods would be checkout



Why use objects?

- Modularity :
  - Source code for an object can be written and maintained independently of the source code for other objects
    - i.e., one file for a car, one file for a plane, etc…
- Code Re-use :
  - Object can be reused in different programs
    - i.e., once your write the code for a car once, you are use it in as many programs as you would like and create as many car objects as you would like.

Outline

- Classes and Objects Revisited
- Using existing Classes
- Anatomy of a Class
- Encapsulation
- Anatomy of a Method
- Method Overloading
- Static Class Members

The First C# Program in this course

- The fundamental OOP concept illustrated by the program:

  An object-oriented program uses objects.

- This program displays a random number on the screen.

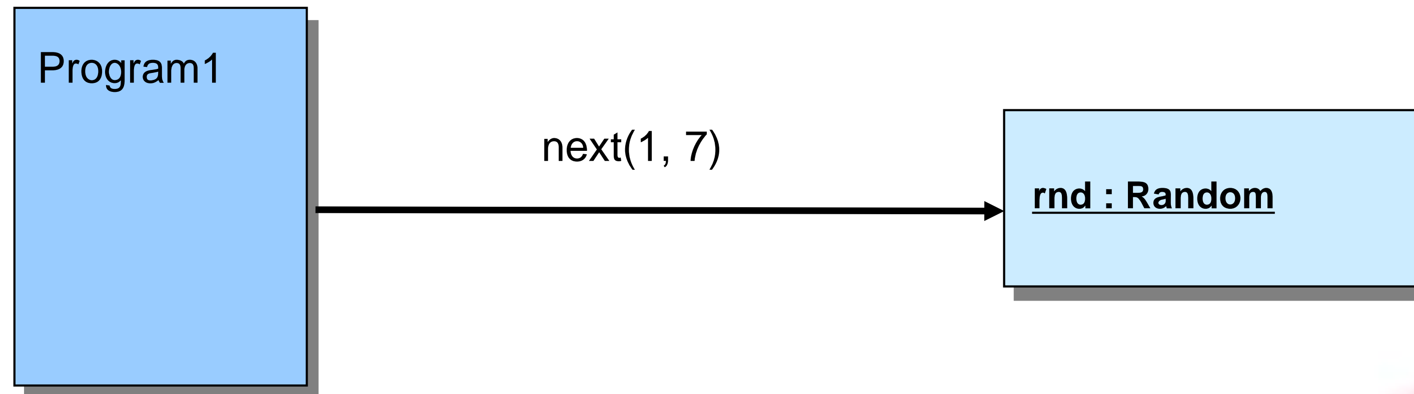
Program1

```
using System;

class Program1
{
    static void Main(string[] args)
    {
        Random rnd = new Random();

        for (int i = 0; i < 100; i++)
        {
            Console.WriteLine(rnd.Next(1, 7);

        }
    }
}
```

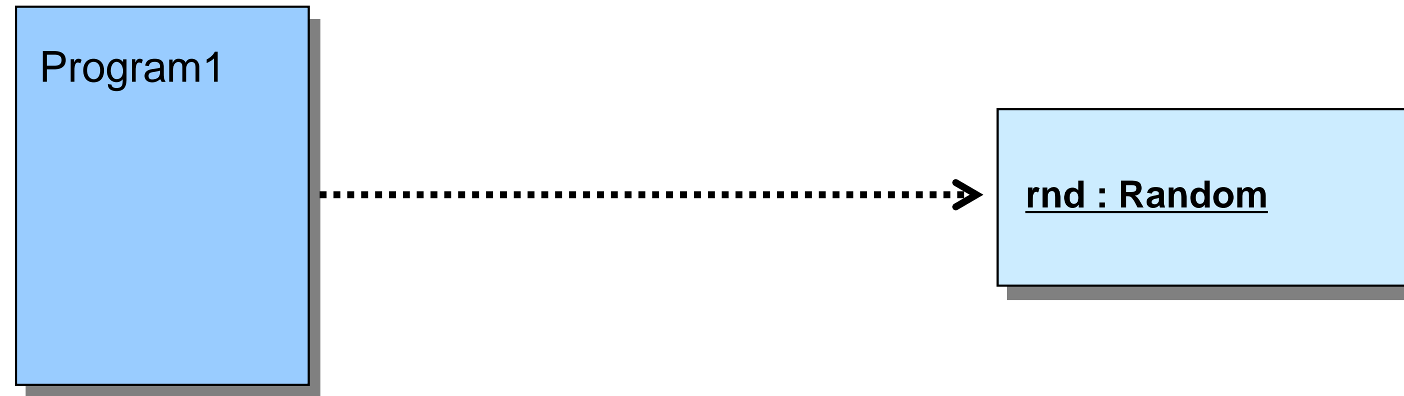
Create an object

Use an object

Program Diagram for program1

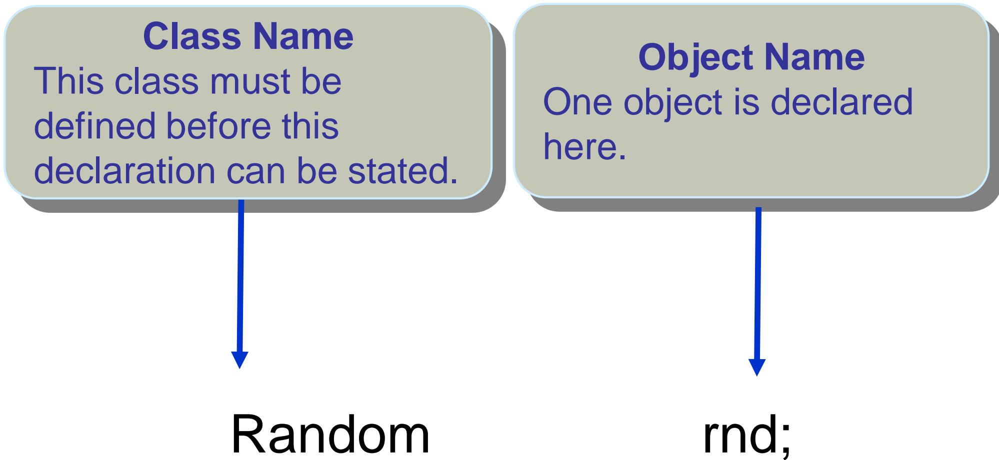


Dependency Relationship



Instead of drawing all messages, we summarize it by showing only the dependency relationship. The diagram shows that Program1 “depends” on the service provided by Random class.

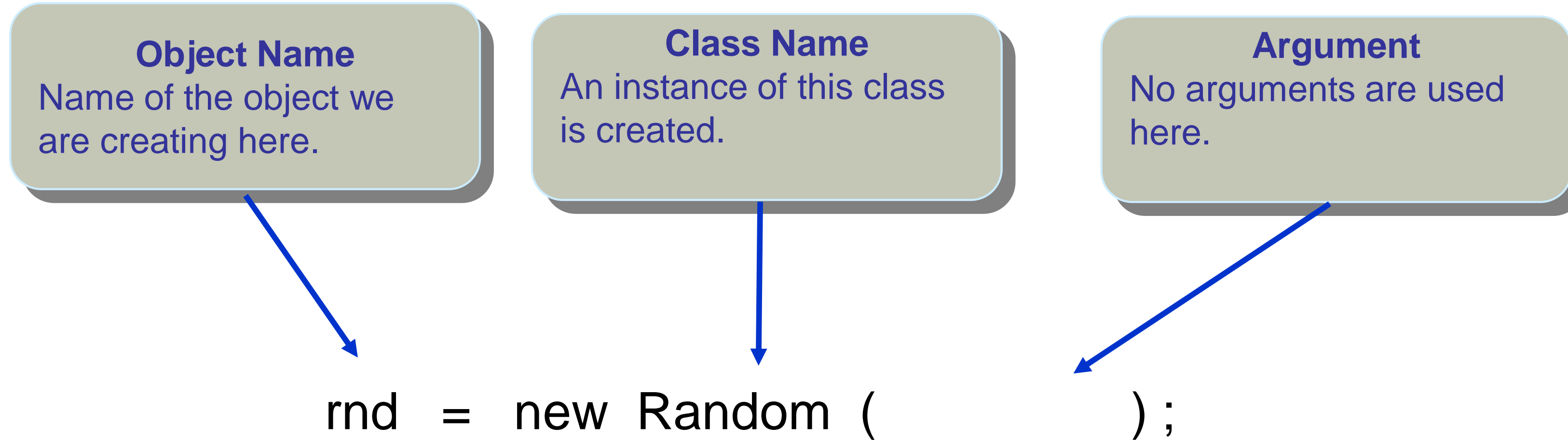
Object Declaration



More Examples

```
Account      customer;
Student      jan, jim, jon;
Vehicle      car1, car2;
```

Object Creation

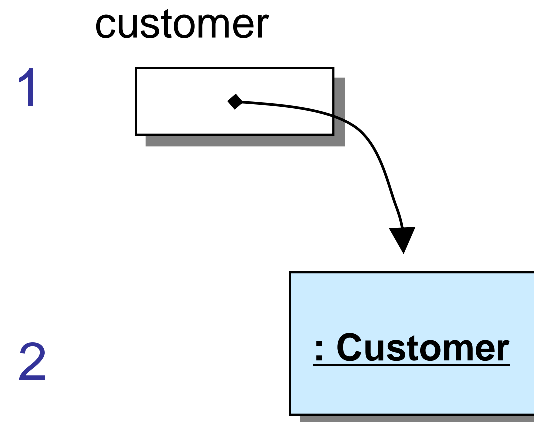


More Examples

```
customer = new Customer( );
jon      = new Student("John C#");
car1     = new Vehicle( );
```

Declaration vs. Creation

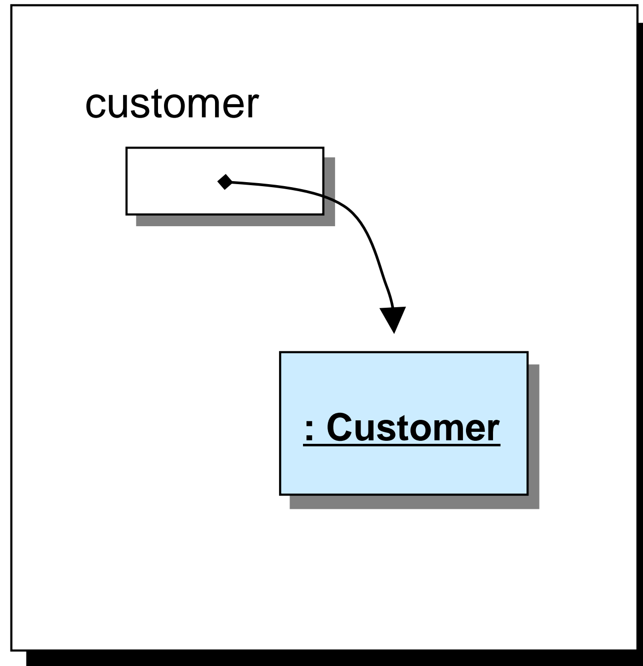
```
1  Customer   customer;
2  customer   =  new  Customer( );
```



1. The identifier customer is declared and space is allocated in memory.

2. A Customer object is created and the identifier customer is set to refer to it.

State-of-Memory vs. Program

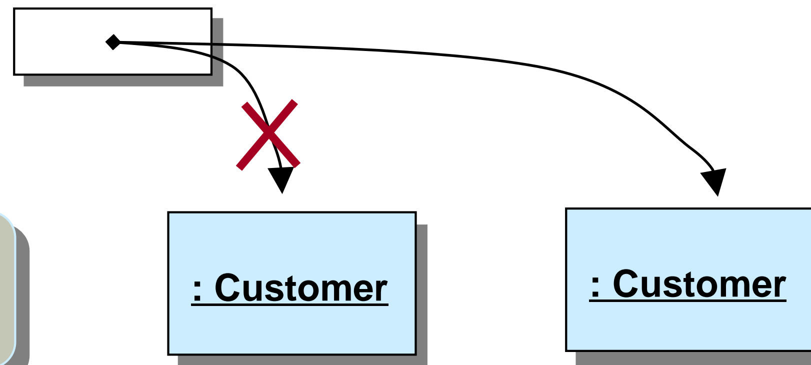


State-of-Memory Notation

Program Diagram Notation

Name vs. Objects

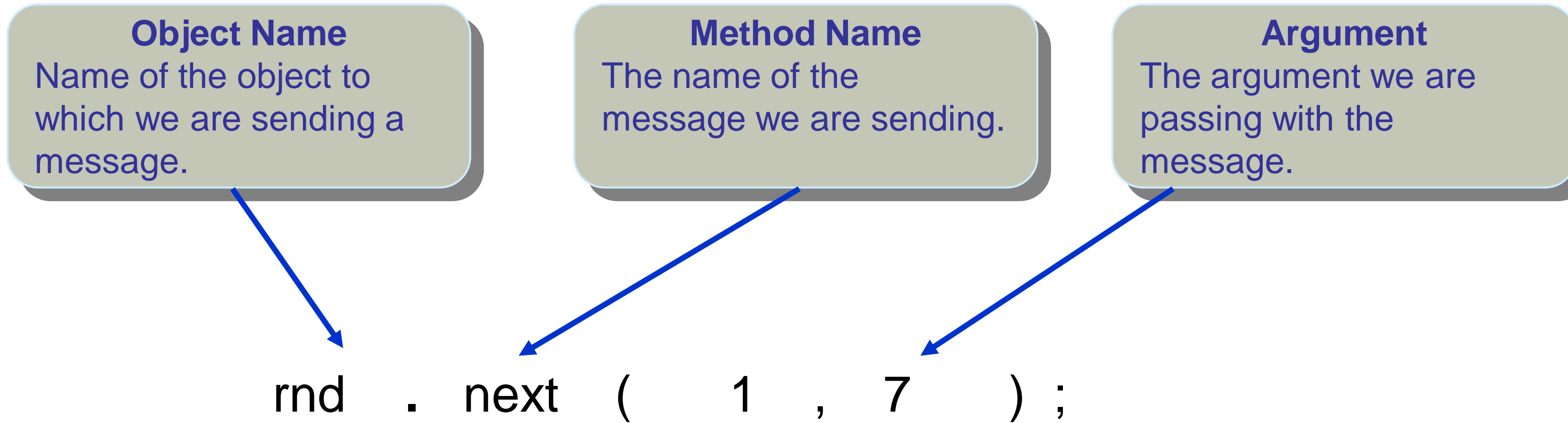
```
Customer    customer;
customer    =   new   Customer( );
customer    =   new   Customer( );
```



Created with the first **new**.

Created with the second **new**. Reference to the first Customer object is lost.

Sending a Message



More Examples

```
account.deposit( 200.0 );
student.setName("john");
car1.startEngine(  );
```

The Die Example

- Consider a six-sided die (singular of dice)
  - It's state can be defined as which face is showing
  - It's primary behavior is that it can be rolled
- We can represent a die in software by designing a class called `Die` that models this state and behavior
- We'll want to design the `Die` class with other data and methods to make it a versatile and reusable resource
- Any given program will not necessarily use all aspects of a given class

Die.C#

```
//*******************************************************************
//  Represents one die (singular of dice) with faces showing values //  between 1 and 6.
//*******************************************************************
using System;
public class DieSample1
      {

         public int faceValue;  // current value showing on the die


         public int roll()
                {
                  Random rnd = new Random();
                  faceValue = rnd.Next(1, 7);
                  return faceValue;
                }
      }
```

The Die Class

- The `DieSample1` class contains two data values
  - an integer `faceValue` that represents the current face value
- The `roll` method uses the `next` method of the `Random` class to determine a new face value

Creating Objects - Instantiation

- The *new* operator creates an instance of a class and reserves memory for it.
- The newly created object is set up by a call to a *constructor* of the Customer class.
- Whenever you use the *new* operator, a special method defined in the given class (a constructor) is called.

```
DieSample1 die1 = new DieSample1();
```

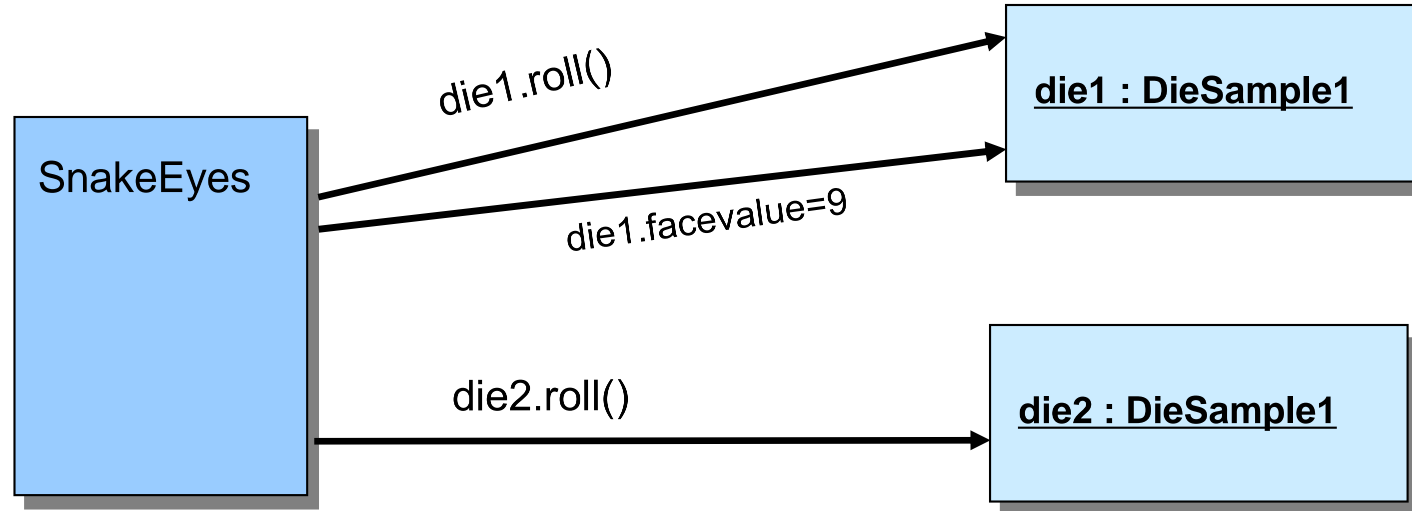
class *variable* *keyword* *constructor*

©2007 · Georges Merx and Ronald J. Norman

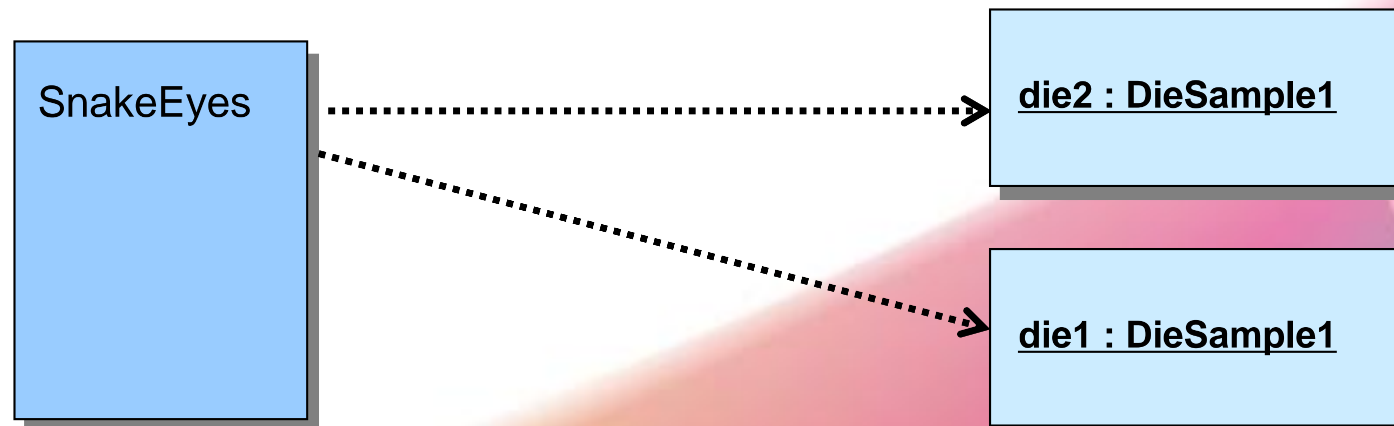
SnakeEyes.C# : Client Application uses two die objects

```
using System;
public class SnakeEyes
{
     //  Creates two Die objects and rolls them several times, counting
     //  the number of snake eyes that occur.
     public static void Main (string [ ] args)
  {
     const int ROLLS = 500;
     int num1, num2, count = 0;

     DieSample1 die1 = new DieSample1();
     DieSample1 die2 = new DieSample1();

     for (int roll=1; roll <= ROLLS; roll++)
            {
               num1 = die1.roll();
               num2 = die2.roll();

               if (num1 == 1 && num2 == 1)    // check for snake eyes
                  count++;
            }
     Console.Writeline ("Number of rolls: " + ROLLS);
     Console.Writeline ("Number of snake eyes: " + count);
     Console.Writeline ("Ratio: " + (float)count / ROLLS);
     die1.facevalue=9;   //wrong value assigned violate encapsulation
  }
}
```

Program Diagram for DieSample1 example



Instead of drawing all messages, we summarize it by showing only the dependency relationship.



Data (Variables) Scope

- **The scope of a variable defines where it can be used in a program.**
- **Data(Variables) declared at the class level can be referenced by all methods in that class**
- **Data declared within a method can be used only in that method**
- **Data declared within a method is called *local data (Variables)***

Scope : Global Variables(instance variables or field variables or data member): Example

Global variables are defined <u>inside the class</u> statement, but <u>outside of the method</u> declaration

Data declared at the class level can be <u>referenced by all methods in that class</u>

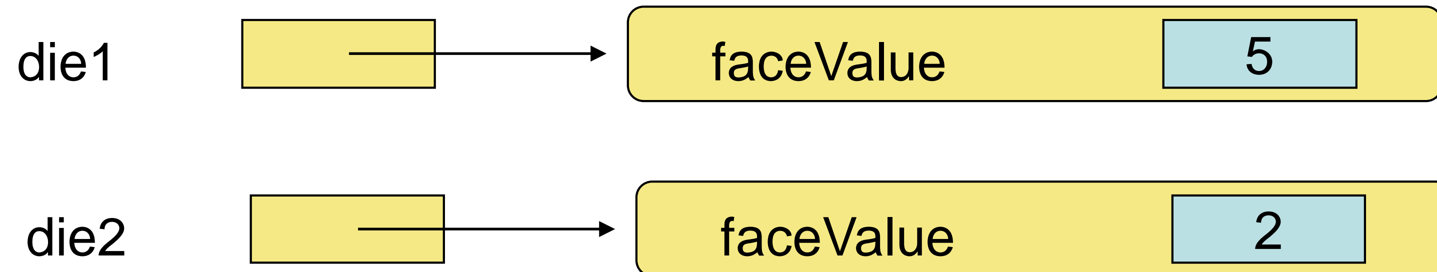
```
public class test
{
    int a;   //This int is GLOBAL and can be
             //used in any method of the class

    public static void main (String args[])
    {
            a = 10; //We used the global int a
                    //inside the code without
                    //having to declare it again
    }
}
```

Instance Data

- The `faceValue` variable in the `Die` class is called *instance data* because each instance (object) that is created has its own version of it
- A class declares the type of the data, but it does not reserve any memory space for it
- Every time a `Die` object is created, a new `faceValue` variable is created as well
- The objects of a class share the method definitions, but each object has its own data space
- That's the only way two objects can have different states

Instance Data

- We can depict the two `Die` objects from the `SnakeEyes` program as follows



Each object maintains its own faceValue variable, and thus its own state

Scope : Methods

- The scope of a variable defines where it can be used in a program.
- As we've seen, local variables can be declared inside a method
- The formal parameters of a method create *automatic local variables* when the method is invoked
- When the method finishes, all local variables are destroyed (including the formal parameters)
- A local variable (the variables we have been creating in our code) can only be used in the method that declares our variable.
- For example : variables declared in the main method are LOCAL to the main method. They can not be used outside of the main method.
  - The only way to use them in another method is to pass them as a parameter.

Scope : For Loops

- A variable defined in the declaration of a “for loop” is local to that loop

- Example :

```
for (int x = 0; x < 10; x++)
{       }
```

- The variable (as defined in the loop) only exists inside of the loop. You can have an entirely different “x” variable outside of this loop.

INSTANCE VARIABLE
VERSUS
LOCAL VARIABLE

INSTANCE VARIABLE	LOCAL VARIABLE
A variable that is bounded to the object itself	A variable that is typically used in a method or a constructor
It is possible to use access modifiers for the instance variables	It is not possible to use access modifiers for the local variables
Can have default values	Do not have default values
Instance variables create when creating an object	Local variables create when entering the method or a constructor
Instance variables destroy when destroying the object	Local variables destroy when exiting the method or a constructor

Visit www.PEDIAA.com

```java
public class A {
    int instanceVariable;
    public void foo() {
        int localvariable;
    }
}
```

Outline

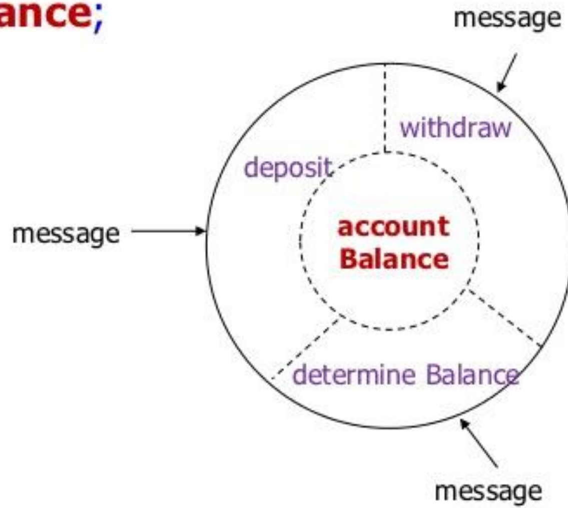
- Classes and Objects Revisited
- Anatomy of a Class
- Encapsulation
- Anatomy of a Method
- Method Overloading
- Static Class Members

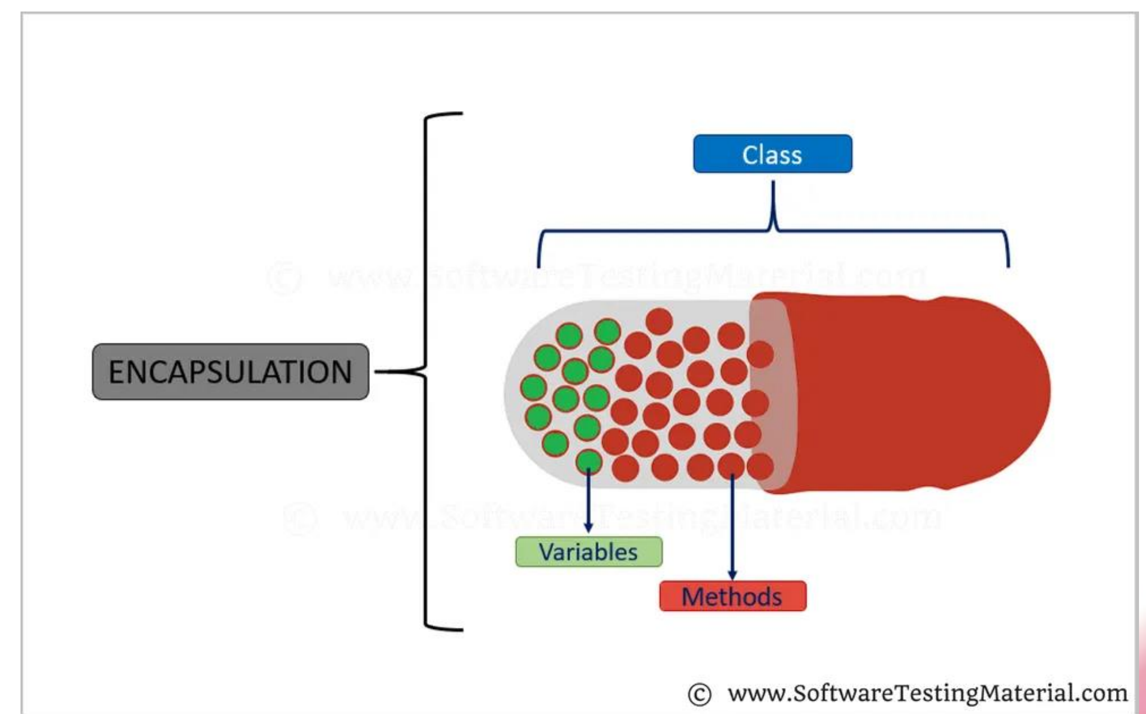
Encapsulation

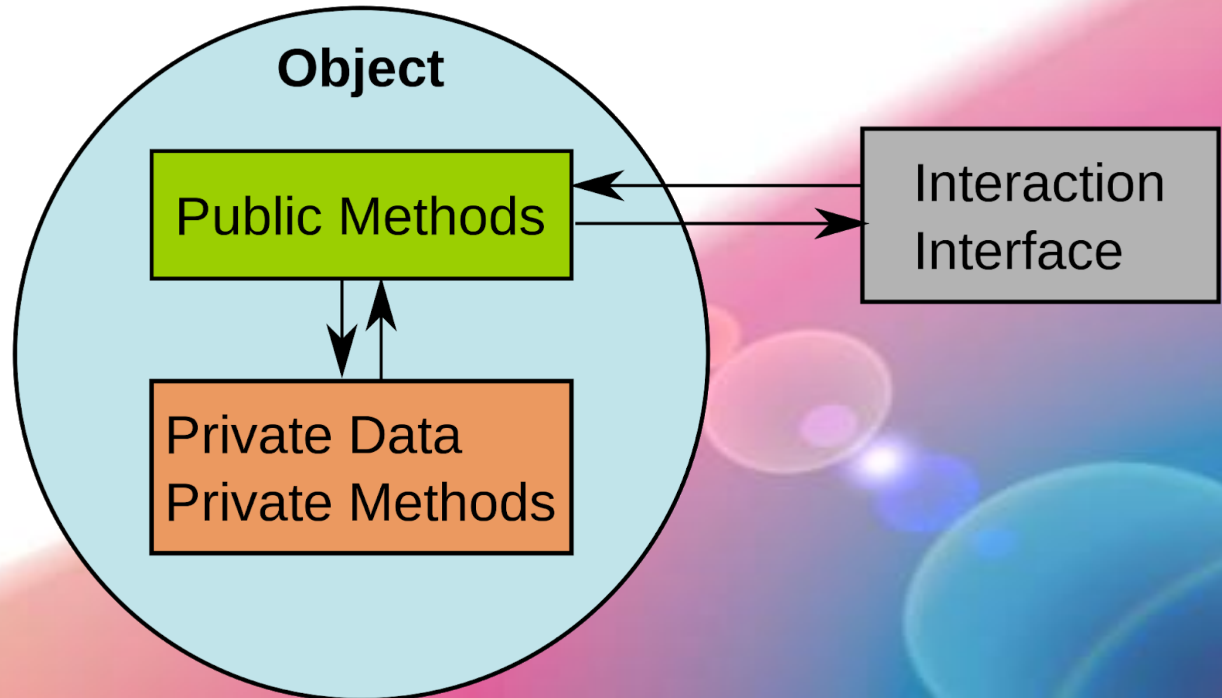
- We can take one of two views of an object
  - internal - the details of the variables and methods of the class that defines it
  - external - the services that an object provides and how the object interacts with the rest of the system
- From the external view, an object is an *encapsulated* entity, providing a set of specific services
- These services define the *interface* to the object

Encapsulation - Example

```
class Account {
   private double accountBalance;

   public withdraw();
   public  deposit();
   public  determineBalance();

} // Class Account
```





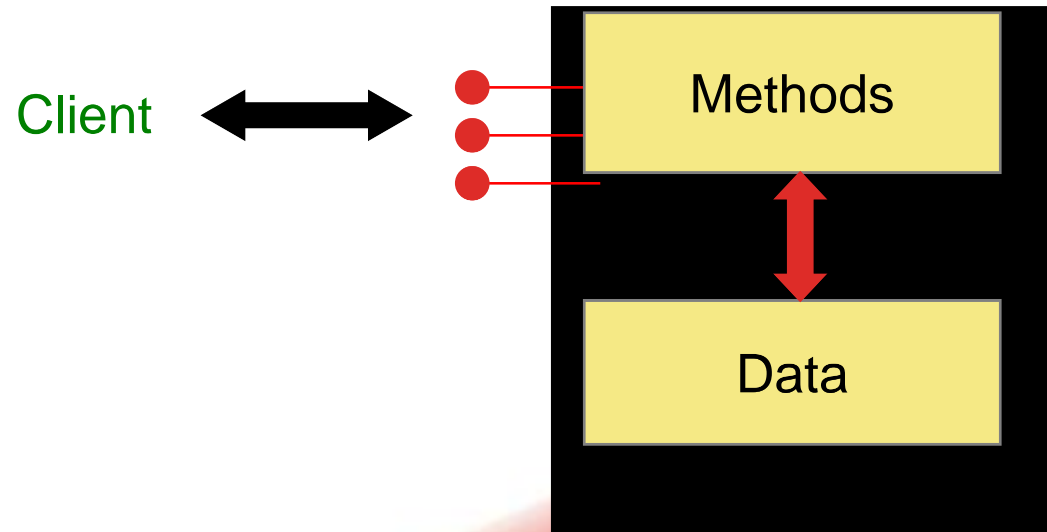


Encapsulation

- One object (called the *client*) may use another object for the services it provides
- The client of an object may request its services (call its methods), but it should not have to be aware of how those services are accomplished
- Any changes to the object's state (its variables) should be made by that object's methods
- We should make it difficult, if not impossible, for a client to access an object's variables directly
- That is, an object should be *self-governing*

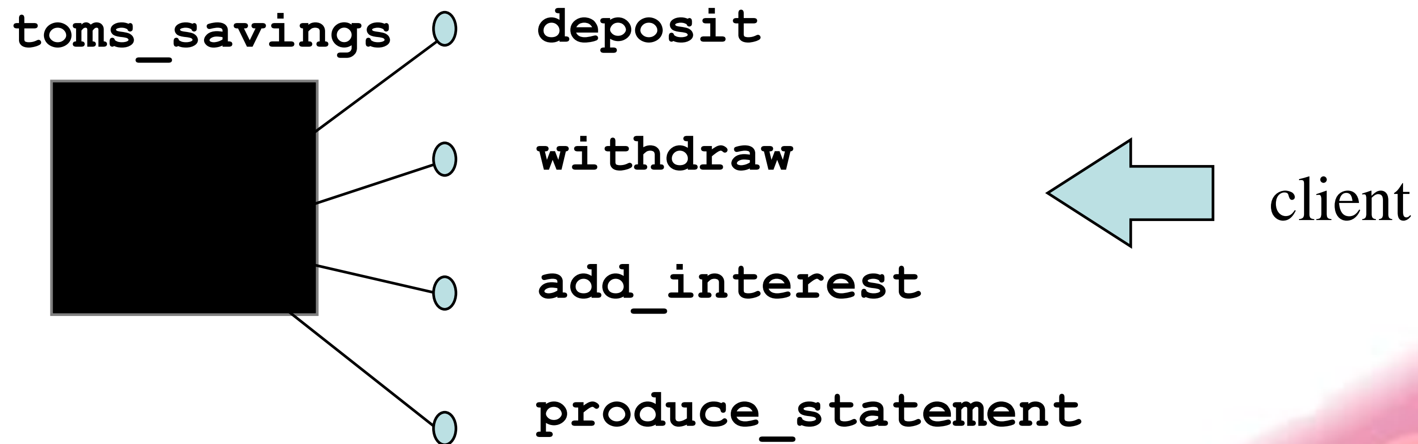
Encapsulation

- An encapsulated object can be thought of as a *black box* – its inner workings are hidden(Abstracted) from the client
- The client invokes the interface methods of the object, which manages the instance data



Encapsulation

- An encapsulated object can be thought of as a *black box*; its inner workings are hidden(Abstracted) to the client



Visibility Modifiers

- In C#, we accomplish encapsulation through the appropriate use of *visibility modifiers*
- A *modifier* is a C# reserved word that specifies particular characteristics of a method or data
- We've used the `const` modifier to define constants
- C# has five visibility modifiers: `public`, `internal`, `private`, `protected` and `protected internal`
- The `protected` modifier involves inheritance, which we will discuss later

Visibility modifie

visibility keyword	Containing Classes	Derived Classes	Containing Assembly	Anywhere outside the containing assembly
public	yes	yes	yes	yes
protected internal	yes	yes	yes	no
protected	yes	yes	no	no
private	yes	no	no	no
internal	yes	no	yes	no

Visibility Modifiers

- **Members of a class that are declared with *public visibility* can be referenced from anywhere**
- **Members of a class that are declared with *private visibility* can be referenced only within that class**
- **Members declared with an internal visibility modifier can be referenced only from within the current project**
- When a class or a class member does not specify a modifier, the default accessibility level of private is assumed.

Visibility Modifiers

- Public variables violate encapsulation because they allow the client to “reach in” and modify the values directly
- Therefore instance variables should not be declared with public visibility
- It is acceptable to give a constant public visibility, which allows it to be used outside of the class
- Public constants do not violate encapsulation because, although the client can access it, its value cannot be changed

Visibility Modifiers

- Methods that provide the object's services are declared with public visibility so that they can be invoked by clients
- Public methods are also called *service methods*
- A method created simply to assist a service method is called a *support method*
- Since a support method is not intended to be called by a client, it should not be declared with public visibility

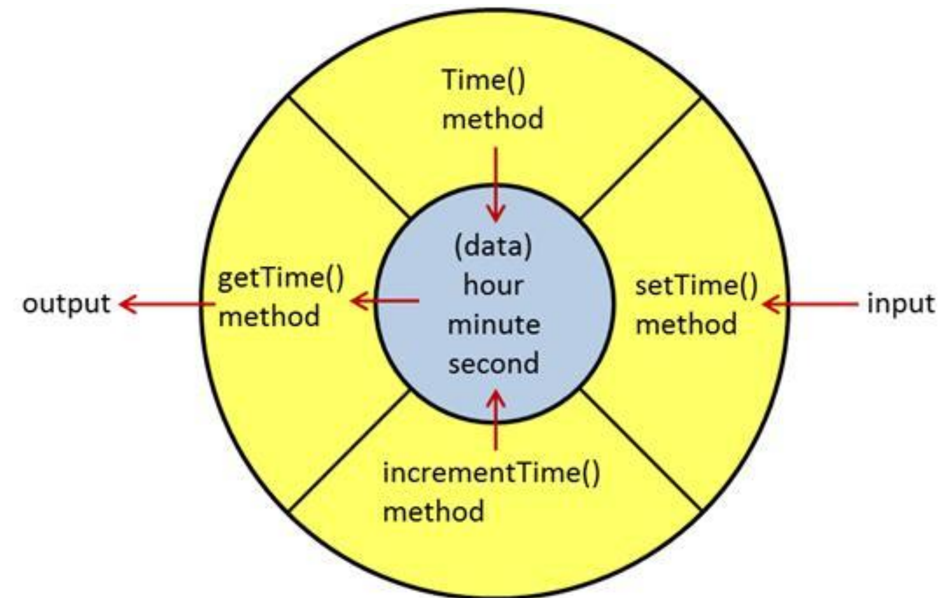
Visibility Modifiers

	public	private
Variables	Violate encapsulation	Enforce encapsulation
Methods	Provide services to clients	Support other methods in the class

DATA ABSTRACTION
VERSUS
ENCAPSULATION

DATA ABSTRACTION	ENCAPSULATION
OOP concept that hides the implementation details and shows only the functionality to the user	OOP concept that binds or wraps the data and methods together into a single unit
Hides the implementation details to reduce the code complexity	Hides data for the purpose of data protection
OOP languages use abstract classes and interfaces to achieve Data Abstraction	OOP languages can achieve Encapsulation by making the data members private and accessing them through public methods

Visit www.PEDIAA.com



Accessors and Mutators

- Because instance data is private, a class usually provides services to access and modify data values
- An *accessor method* returns the current value of a variable
- A *mutator method* changes the value of a variable
- The names of accessor and mutator methods take the form `getX` and `setX`, respectively, where `X` is the name of the value

Accessors and Mutators

- They are sometimes called "getters" and "setters"
- Coin.cs
  - The getFaceValue method is an accessor
  - The setFaceValue method is a mutator

Die.cs Encapsulated

```
//********************************************************************
//  Die2.cs
//  Represents one die (singular of dice) with faces showing values
//  between 1 and 6.
//********************************************************************

using System;
public class Die2
{
   private const int MAX = 6;  // maximum face value

   private int faceValue;  // current value showing on the die


   public int roll()
   {
     Random rnd = new Random();
     faceValue = rnd.Next(1, 7);
     return faceValue;
   }
}
```

Die.cs Encapsulated

```
   //-----------------------------------------------------------------
   //  Face value mutator. The face value is not modified if the
   //  specified value is not valid.
   //-----------------------------------------------------------------
   public void setFaceValue (int value)
   {
      if (value > 0 && value <= MAX)
         faceValue = value;
   }
   //-----------------------------------------------------------------
   //  Face value accessor.
   //-----------------------------------------------------------------
   public int getFaceValue()
   {
      return faceValue;
   }
}
```

Mutator Restrictions

- The use of mutators gives the class designer the ability to restrict a client's options to modify an object's state
- A mutator is often designed so that the values of variables can be set only within particular limits
- For example, the `setFaceValue` mutator of the `Die` class should have restricted the value to the valid range (1 to `MAX`)
- Such restrictions can be implemented through the use of an `if` statement in the body of the constructor.

Creating Objects - Instantiation

- The *new* operator creates an instance of a class and reserves memory for it.
- The newly created object is set up by a call to a *constructor* of the Die class.
- Whenever you use the *new* operator, a special method defined in the given class (a constructor) is called.

```
Die die1 = new Die();
```

class — *variable* — *keyword* — *constructor*

Constructor

- A ***constructor*** is a special method that is executed when a new instance of the class is created.

```
public <class name> ( <parameters> ){
    <statements>
}
```



```
public    Bicycle      (      ) {
    ownerName = "Unassigned";
}
```

- A *constructor* is a special method that is used to set up an object when it is initially created
- A constructor has the same name as the class

Constructors

- Is a special method that is used to set up a newly created object.
- Often sets the initial values of variables allocates memory for it.
- It can have parameters, which are often used to initialize some variables in the object.
- Always has the same name as the class.
- Does not return a value.
- Has no return type, not even `void`.
- Called when keyword new is followed by the class name and parentheses
- A constructor with an empty parameter list is called a *default constructor*.
  - This is included (by default) by the compiler in any class that does not include its own constructor
  - It is NOT included if there is ANY constructor defined in the class. In other words, you would then have to create your own default constructor if you wanted to have one.

Die.cs Encapsulated + Constructor

```
//*******************************************************************
//  Die3.cs       //
//  Represents one die (singular of dice) with faces showing values
//  between 1 and 6.
//*******************************************************************
using System;
public class Die3
{

   private int faceValue;  // current value showing on the die

   //-----------------------------------------------------------------
   //  Constructor: Sets the initial face value of this die.
   //-----------------------------------------------------------------
   public Die3()
   {
      faceValue = 1;
   }
```

(more…)

Die.cs Encapsulated + Constructor

```
    //----------------------------------------------------------------
   //  Computes a new face value for this die and returns the result.
   //----------------------------------------------------------------
   public int roll()
   {
     Random rnd = new Random();
     faceValue = rnd.Next(1, 7);

      return faceValue;
   }

   //----------------------------------------------------------------
   //  Face value mutator. The face value is not modified if the
   //  specified value is not valid.
   //----------------------------------------------------------------
   public void setFaceValue (int value)
   {
      if (value > 0 && value <= MAX)
         faceValue = value;
   }
   //----------------------------------------------------------------
   //  Face value accessor.
   //----------------------------------------------------------------
   public int getFaceValue()
   {
      return faceValue;
   }
}
```

Property

- A property is a member that provides a flexible mechanism to read, write, or compute the value of a private field.
-
- Properties can be used as if they are public data members, but they are actually special methods called accessors.

- This enables data to be accessed easily and still helps promote the safety and flexibility of methods

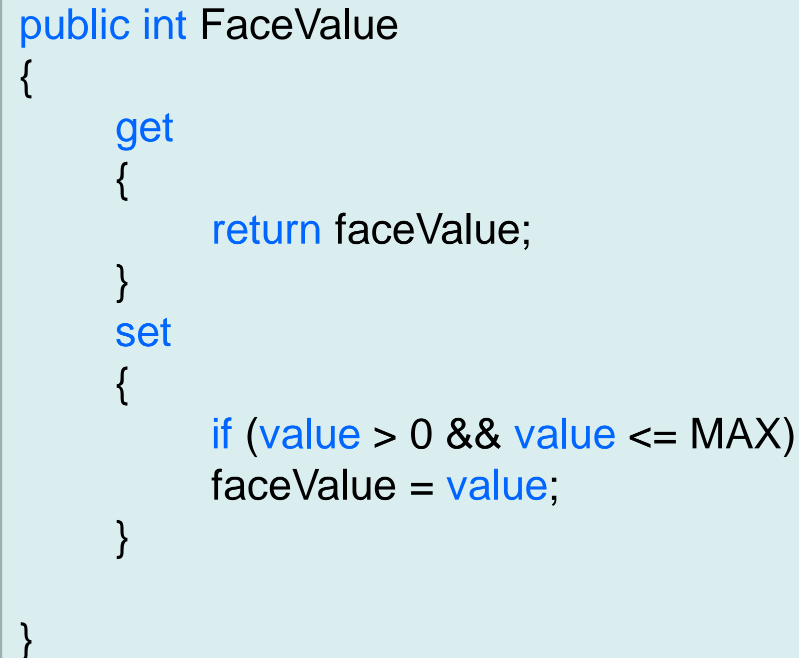
- Many languages provide mutators, or setters that enable the data to be changed and accessors or getters that enable the data to be retrieved C# introduced properties.
- A property looks like a data field, but it does not directly represent a storage location.
- Properties are more closely aligned to methods. They provide a way to change or retrieve private member data

- Standard naming convention in C# for properties: Use the same name as the instance variable or field, but start with uppercase character

Die.cs Encapsulated + Constructor

```
//-----------------------------------------------------------------
//  Computes a new face value for this die and returns the result.
//-----------------------------------------------------------------
public int roll()
{
  Random rnd = new Random();
  faceValue = rnd.Next(1, 7);

   return faceValue;
}

//-----------------------------------------------------------------
//  Face value mutator. The face value is not
// modified if the specified value is not valid.
//-----------------------------------------------------------------
public void setFaceValue (int value)
{
   if (value > 0 && value <= MAX)
      faceValue = value;
}
//-----------------------------------------------------------------
//  Face value accessor.
//-----------------------------------------------------------------
public int getFaceValue()
{
   return faceValue;
}
}
```

- Properties are more closely aligned to methods. They provide a way to change or retrieve private member data
- Standard naming convention in C# for properties: Use the same name as the instance variable or field, but start with uppercase character



```
public int FaceValue
{
        get
        {
                return faceValue;
        }
        set
        {
                if (value > 0 && value <= MAX)
                faceValue = value;
        }

}
```

Accessing private fields using

get and set methods

Read:

```
Console.WriteLine(die1.GetFaceValue());
```

Write:

```
die1.SetFaceValue(5);
```

properties

Read:

```
Console.WriteLine(die1.FaceValue);
```

Write:

```
die1.FaceValue=5;
```

I tend to use properties if the following are true:

- The property will return a single, logic value
- Little or no logic is involved (typically just return a value, or do a small check/return value)

I tend to use methods if the following are true:

- There is going to be significant work involved in returning the value - ie: it'll get fetched from a DB, or something that may take "time"
- There is quite a bit of logic involved, either in getting or setting the value

Creating Classes

- **The syntax for defining a class is:**

```
class class-name //class declaration
 {
   declarations of instance variables
    constructors
    service methods
    support methods          //methods        // class body
    getters and setters
    Property
 }
```

- **The class declaration declares the name of the class along with other attributes.**
- **The variables, constructors, and methods of a class are generically called *members* of the class.**

Bank Account Example

- Let’s look at another example that demonstrates the implementation details of classes and methods
- We’ll represent a bank account by a class named `Account`
- It’s state can include the account number, the current balance, and the name of the owner
- An account’s behaviors (or services) include deposits and withdrawals, and adding interest

Driver Programs

- **A *driver program* drives the use of other, more interesting parts of a program**
- **Driver programs are often used to test other parts of the software**
- **The `Transactions` class contains a `main` method that drives the use of the `Account` class, exercising its services**

Transactions.cs

```
//*********************************************************************
//  Transactions.cs
//
//  Demonstrates the creation and use of multiple Account objects.
//*********************************************************************

public class Transactions
{
   //-----------------------------------------------------------------
   //  Creates some bank accounts and requests various services.
   //-----------------------------------------------------------------
   public static void Main (string[] args)
   {
      Account acct1 = new Account ("Ted Murphy", 72354, 25.59);
      Account acct2 = new Account ("Angelica Adams", 69713, 500.00);
      Account acct3 = new Account ("Edward Demsey", 93757, 769.32);

      acct1.Deposit (44.10);  // return value ignored

      double adamsBalance = acct2.Deposit (75.25);
      Console.Writeline ("Adams balance after deposit: " +
                         adamsBalance);

(more…)
```

Transactions.cs

```
    Console.Writeline ("Adams balance after withdrawal: " +
                       acct2.Withdraw (480, 1.50));

    acct3.Withdraw (-100.00, 1.50);  // invalid transaction

    acct1.AddInterest();
    acct2.AddInterest();
    acct3.AddInterest();

    Console.Writeline ();
    Console.WriteLine(acct1.getAcountInfo());
    Console.WriteLine(acct2.getAcountInfo());
    Console.WriteLine(acct3.getAcountInfo());   }
}
```

Account.cs

```
//********************************************************************
//  Account.cs       C# Foundations
//
//  Represents a bank account with basic services such as deposit
//  and withdraw.
//********************************************************************


public class Account
{
   private const double RATE = 0.035;  // interest rate of 3.5%

   private string name;
   private long acctNumber;
   private double balance;

(more…)
```

Account.cs

```
//-----------------------------------------------------------------
//  Sets up this account with the specified owner, account number,
//  and initial balance.
//-----------------------------------------------------------------
public Account (string owner, long account, double initial)
{
   name = owner;
   acctNumber = account;
   balance = initial;
}

//-----------------------------------------------------------------
//  Deposits the specified amount into this account and returns
//  the new balance. The balance is not modified if the deposit
//  amount is invalid.
//-----------------------------------------------------------------
public double Deposit (double amount)
{
   if (amount > 0)
      balance = balance + amount;

   return balance;
}
```

(more…)

Account.cs

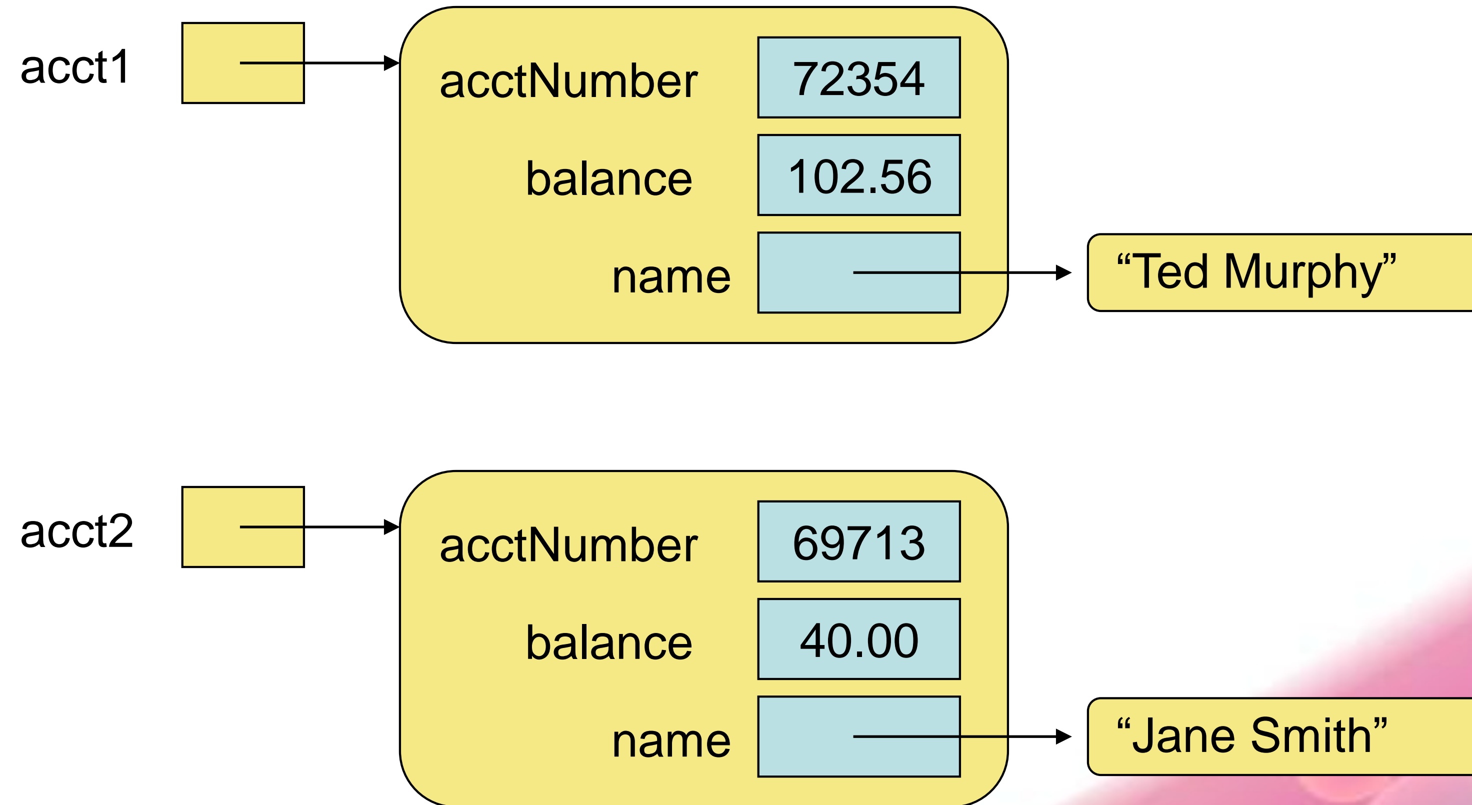
```
//-----------------------------------------------------------------
//  Withdraws the specified amount and fee from this account and
//  returns the new balance. The balance is not modified if the
//  withdraw amount is invalid or the balance is insufficient.
//-----------------------------------------------------------------
public double Withdraw (double amount, double fee)
{
   if (amount+fee > 0 && amount+fee < balance)
      balance = balance - amount - fee;

   return balance;
}

//-----------------------------------------------------------------
//  Adds interest to this account and returns the new balance.
//-----------------------------------------------------------------
public double AddInterest ()
{
   balance += (balance * RATE);
   return balance;
}
```

(more…)

Account.cs

```
   //----------------------------------------------------------------
   //  Returns the current balance of this account.
   //----------------------------------------------------------------
   public double GetBalance ()
   {
      return balance;
   }

   //----------------------------------------------------------------
   //  Returns a one-line description of this account as a string.
   //----------------------------------------------------------------
   public String  getAcountInfo()
      {

         return ("The acount number: "+ acctNumber +"\nCoustomer Name:" + name +
                  "\n The balance is: "+ balance);
      }

}
```

Bank Account Example



Bank Account Example

- There are some improvements that can be made to the `Account` class
- Formal getters and setters could have been defined for all data
- The design of some methods could also be more robust, such as verifying that the `amount` parameter to the `withdraw` method is positive

Outline

- Classes and Objects Revisited
- Anatomy of a Class
- Encapsulation
- Method Overloading
- Static Class Members

Overloaded Methods

- **Methods can share the same name as long as**
  - **they have a different number of parameters (Rule 1) or**
  - **their parameters are of different data types when the number of parameters is the same (Rule 2)**

```
public void myMethod(int x, int y) { ... }
public void myMethod(int x) { ... }
```

√ Rule 1

```
public void myMethod(double x) { ... }
public void myMethod(int x) { ... }
```

√ Rule 2

Overloaded Constructor

- The same rules apply for overloaded constructors
  - this is how we can define more than one constructor to a class

```
public Person( ) { ... }
public Person(int age) { ... }
```

✓ Rule 1

```
public Pet(int age) { ... }
public Pet(String name) { ... }
```

✓ Rule 2

©The McGraw-Hill Companies, Inc. Permission required for reproduction or display.

Method Overloading

- If a method is overloaded, the method name is not sufficient to determine which method is being called
- The *signature* of each overloaded method must be unique
- The signature includes the number, type, and order of the parameters
- Examples
  - int MyMethod(int x)
  - int MyMethod(double y)
  - int MyMethod(int a, double b)

Method Overloading

- **The compiler determines which method is being invoked by analyzing the parameters**

```
float tryMe(int x)
{
  return x + .375;
}
```

Invocation

```
result = tryMe(25, 4.32)
```

```
float tryMe(int x, float y)
{
  return x * y;
}
```

Methods and the Compiler

- How does the compiler know which method is the correct one to use?
  - When compiling, the compiler first checks the method name.
  - If multiple methods with the same name exist, it will then go to the parameter list to decide which method is the correct one.
  - If there are two (or more) methods with the same name and same parameter list, the compiler will return an error.
- Is this correct?
  - int MyMethod(int x)
  - double MyMethod(int a)

  - Are these two methods allowed together?

Is this correct? NO

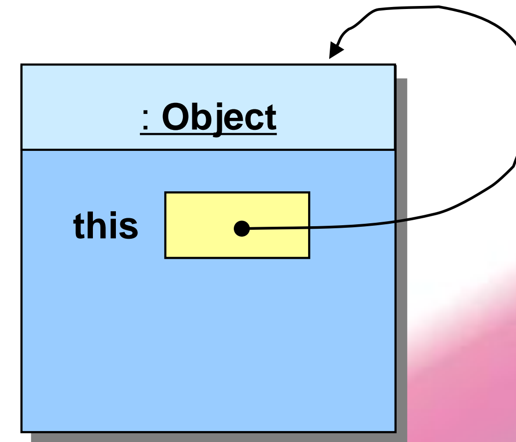
- int MyMethod(int x)
- double MyMethod(int a)
- The compiler would not know which of the two methods you wish to call.
- The compiler chooses the method based on the method name first and then the parameter list (it does not care about the return type).

Overloading Methods

- The return type of the method is <u>not</u> part of the signature
- That is, overloaded methods cannot differ only by their return type
- Constructors can be overloaded
- Overloaded constructors provide multiple ways to initialize a new object

Reserved Word this

- The reserved word this is called a *self-referencing pointer* because it refers to an object from the object's method.



- That is, the this reference, used inside a method, refers to the object through which the method is being executed
- Can also be used to call one constructor for another in a class …. (see next slide)

The keyword *this* : calling one constuctor from another

```
// constructor with 3 params
public Date(int m, int d, int y)
{
    month = m; day = d; year = y;
}

// constructor with 2 params
public Date(int m, int d)
{
     : this(m,d,0); //calls constructor with 3 params
}
```

The keyword *this* : as a reference variable

```
public class Date
{
     private int m, d, y;


    // constructor with 3 params
    public Date(int m, int d, int y)
      {
       this.m = m;   (Set this object's m instance variable)
       this.d = d;   (Set this object's d instance variable)
       this.y = y;   (Set this object's y instance variable)
      }
}
```

Outline

- Classes and Objects Revisited
- Anatomy of a Class
- Encapsulation
- Anatomy of a Method
- Method Overloading
- Static Class Members

Static Class Members

- Each object has its own copy of all the instance variables of the class.
- Sometimes, only a single copy of a particular variable should be shared by all objects created from a class.
- A *static field* (or *class variable*) is used in this case.
- Represents classwise information
  - all objects in the class share the same piece of information

Keyword : static

- Used to declare a static variable
- When to use static?
  - Sometimes, you want to know how many objects from a particular class exist
  - You can create a static “count” variable in the class definition --- and have that variable incremented/decremented upon object initialization/deinitialization.

Static

- Static variables have class scope
- Static class members exist even when no objects of the class exist
- They are available as soon as the class is loaded into memory at execution time.
    - To access a public static member when no objects exist, prefix the class name and a dot (.) to the to the static member (such as Math.PI).
    - To access private static members when no object exists, there must be a public static method provided and the method must be called by qualifying its name with the class name and a dot.

Static : Methods

- A method declared static cannot acces non-static class members because a static method can be called even when no objects of the class have been instantiated.

- Additionally, the “this” reference can not be used in a static method (same reason as above).

- Static methods and static variables often work together

- The following example keeps track of how many `Slogan` objects have been created using a static variable, and makes that information available using a static method

Slogan.cs

```
//*********************************************************************
//  Slogan.cs
//
//  Represents a single slogan or motto.
//*********************************************************************

public class Slogan
{
   private String phrase;
   private static int count = 0;

   //-----------------------------------------------------------------
   //  Constructor: Sets up the slogan and increments the number of
   //  instances created.
   //-----------------------------------------------------------------
   public Slogan (String str)
   {
      phrase = str;
      count++;
   }

(more…)
```

Slogan.cs

```
   //-----------------------------------------------------------------
   //  Returns this slogan as a string.
   //-----------------------------------------------------------------
   public String GetString()
   {
      return phrase;
   }

   //-----------------------------------------------------------------
   //  Returns the number of instances of this class that have been
   //  created.
   //-----------------------------------------------------------------
   public static int GetCount ()
   {
      return count;
   }
}
```

SloganCounter.cs

```
//********************************************************************
//  SloganCounter.cs       //
//  Demonstrates the use of the static modifier.
//********************************************************************
using System;
public class SloganCounter
{
   //-----------------------------------------------------------------
   //  Creates several Slogan objects and prints the number of
   //  objects that were created.
   //-----------------------------------------------------------------
   public static void Main (string[] args)
   {
      Slogan obj;

      obj = new Slogan ("Remember the Alamo.");
     Console.WriteLine (obj.GetString());

      obj = new Slogan ("Don't Worry. Be Happy.");
      Console.WriteLine (obj.GetString());

      obj = new Slogan ("Live Free or Die.");
      Console.WriteLine(obj.GetString());
```

(more…)

SloganCounter.cs

```
        obj = new Slogan ("Talk is Cheap.");
        Console.WriteLine (obj.GetString());

        obj = new Slogan ("Write Once, Run Anywhere.");
        Console.WriteLine (obj.GetString());

        Console.WriteLine();
        Console.WriteLine ("Slogans created: " + Slogan.GetCount());
    }
}
```