# Part 7 - Object Oriented Concepts
Classes, Objects, Properties and Methods

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object Orientated Paradigm</td>
<td>2</td>
</tr>
<tr>
<td>Abstract Data Types (ADT) &amp; Encapsulation</td>
<td>3</td>
</tr>
<tr>
<td>Encapsulation</td>
<td>5</td>
</tr>
<tr>
<td>Classes and Objects</td>
<td>6</td>
</tr>
<tr>
<td>Methods and Messages</td>
<td>9</td>
</tr>
<tr>
<td>Inheritance and Multiple Inheritance</td>
<td>10</td>
</tr>
<tr>
<td>Specialization</td>
<td>12</td>
</tr>
<tr>
<td>Extendibility</td>
<td>12</td>
</tr>
<tr>
<td>Polymorphism</td>
<td>12</td>
</tr>
<tr>
<td>Polymorphism and Overloading</td>
<td>13</td>
</tr>
<tr>
<td>Dynamic or Late Binding (Overriding)</td>
<td>14</td>
</tr>
</tbody>
</table>
MODELING THE REAL WORLD

Humans natural ability to classify, generalize and create abstractions in order to model the world.

View the world as a series of entities and the interaction among them.

Problems are solved through the interaction among a number of autonomous and cooperative objects.
Abstract Data Types (ADT) & Encapsulation

! View the **Code and Data** as an **atomic and non-decomposable unit**.

! Traditional programming languages deliver modularization by the use of procedures and functions.

! Object oriented languages usually implement modularization through Abstract Data Types (or objects).

! An object is defined as a set of built-in and/or user-defined data types along with a set of operations which manipulate them.

\[
\text{ADT} = \text{Data} + \text{Code}
\]

! The functions which surround the data structure provide a robust interface to the clients of that data.

! Simplifies the **debugging, maintenance** and **testing**.

! The applications or clients which use object will not be effected by the possible changes in the implementation of the object, as long as the interface or behavior of the object remains constant.

! ADT's and Encapsulations, makes large projects intellectually manageable.
Encapsulation

! ADT's provide a "LOGICAL" view for encapsulation, however, in a traditional programming approach one may inadvertently bypass ADT boundaries and access a given data structure, therefore, introducing potential errors.

! Encapsulation provides the "PHYSICAL" method for ensuring ADT boundaries are not breached, therefore, preventing the client from unauthorized access.

! Encapsulation is achieved by tightly coupling the data structure and its related methods (functions) hence viewing them as an atomic unit.

! Access to the data is only possible through the public interface provided. Encapsulation hides the details of the object from the client.
Definitions of a Class:

1) A set of similar objects.

2) The term used to define abstract data types in object oriented programming languages.

A class provides the mechanism for syntactically expressing the semantics of encapsulation.

```
class CLASS_NAME {
    private variables
    private data structure
    ... ...
    public code
    public method names
    ... ...
}
```
Public Class CAR
    ' Private Data
    Private color As Integer
    Private year As Integer
    Private model As String
    Private make As String
    '-------------------------------
    ' Public Methods
    Public Sub PrintColor() 'return the color attribute
        Console.WriteLine("The Color = {0}", color)
    End Sub
    '-------------------------------
    Public Sub SetColor(ByVal clr As Integer)
        color = clr
    End Sub
    '-------------------------------
End Class

Module Module1
    Sub Main()
        Const BLUE As Integer = 7
        Const BLACK As Integer = 3

        Dim MyCar As CAR
        MyCar = New CAR

        MyCar.SetColor(BLUE)
        MyCar.PrintColor()

        Console.WriteLine(MyCar.color)  ' Note that this is a error, since
                                       ' we don't have direct access to the
                                       ' Color attribute.

        Console.ReadLine()
    End Sub
End Module
Objects and Classes

! An object is an element of a given class. (i.e. A variables of that type)

! An object is also called an instance of the class.

EXAMPLE:

1) "ButtonQuit" is an object of type "Button".

2) "mycar" is an object of type "CAR". One may access the contents of an object (its instance variables) and change the state of this object by invoking or sending message to the object's methods.
The sub-programs (procedures and functions) within the object are called methods.

The process of invoking a method by its name is called sending a message to the object.

The state of an object can be inspected and modified by sending a message to one of its methods.

**EXAMPLE OF A MESSAGE:**

MyCar.SetColor(BLUE)

Sends the message to object "MyCar" to invoke the "SetColor" method.
Inheritance and Multiple Inheritance

! One of the most important aspect of object oriented systems.

! A new class can inherit or reuse the behavior and structure of previously defined classes.

CLASS

X
5
5
5
5
V

SUB-CLASS

! The Sub-Classes (Derived Classes) can extend or add more functionality (i.e. methods and instance variables) to the base class.

EXAMPLE OF A SUPER_CAR:

class SUPER_CAR Inherits From CAR
  private SuperAttribute as String
  public sub GetSuperAttribute(ByVal Clr As Integer)
  public sub SetSuperAttribute(ByVal SuperAttr As String)
End Class

Dim BatMobil As SUPER_CAR ‘An object of the type SUPER_CAR
ONE CAN ACCESS THE INHERITED DATA AND CODE AS FOLLOWS:

BatMobil.SetColor(Black)
BatMobil.SetSuperAttribute("Goes Real Fast!!")

Inheritance is a natural tool to express relationships such as classification, specialization, generalization, evolution and approximation.
Public Class CAR
    ' Private Data
    Private color As Integer
    Private year As Integer
    Private model As String
    Private make As String
    '-----------------------------
    ' Public Methods
    Public Sub PrintColor() 'return the color attribute
        Console.WriteLine("The Color = {0}", color)
    End Sub
    '-----------------------------
    Public Sub SetColor(ByVal clr As Integer)
        color = clr
    End Sub
    '-----------------------------
End Class

' The SUPER_CAR class Inherits from the CAR class
' and add extends the class by adding a new attribute
' and two new methods.

Class SUPER_CAR
    Inherits CAR ' Inherits from the CAR class

    Public SuperAttribute As String
    '-----------------------------
    Public Function GetSuperAttribute() As String
        Return SuperAttribute
    End Function
    '-----------------------------
    Public Sub SetSuperAttribute(ByVal SuperAttr As String)
        SuperAttribute = SuperAttr
    End Sub
End Class
Module Module1

Sub Main()

    Const BLUE As Integer = 7
    Const BLACK As Integer = 3

    Dim MyCar As CAR
    MyCar = New CAR
        ' MyCar is a Reference Variable
        ' The NEW operator will actually
        ' allocate the memory needed for the
car object.

    MyCar.SetColor(BLUE)
    MyCar.PrintColor()

    'Console.WriteLine(MyCar.color)   ' Note that this is a error, since
                                ' we don't have direct access to the
                                ' Color attribute.

    '-------------------------------

    Dim BatMobil As New SUPER_CAR
    BatMobil.SetColor(BLACK)

    BatMobil.SetSuperAttribute("Goes Real Fast!!")
    BatMobil.PrintColor()
    Console.WriteLine(BatMobil.SuperAttribute)   'Note that I can
                                                ' access SuperAttribute
                                                ' because it is public.

    '-------------------------------

    Console.ReadLine()

End Sub

End Module
INHERITANCE DELIVERS THREE IMPORTANT FEATURES:

1) **Specialization**  (deriving from a general class and making it more specific.)

2) **Extendibility**  (extending a system by inheriting classes and adding to their functionality, so called incremental development.)

3) **Polymorphism**  (facilitates polymorphism by allowing functions to be overloaded at different levels of the class hierarchy to achieve different objectives.)
Polymorphism and Overloading

! Polymorphism comes from the Greek: "having many shapes".

! Polymorphism allows the programmer to use the same method name to perform a given generic task on a number of different data types. For example:
   Dim x As Integer
   Dim y As Double
   Dim z As String
   initialize(x)
   initialize(y)
   initialize(z)

! Even though the function names in the above examples are the same, the compiler is able to select the correct version.

! Conflicts are resolved by Static or early binding. During compile time, the compiler will check the type of each argument and determine which function must be called.

! Overloading is not a new concept, most if not all conventional languages provide overloaded operators such as (+, -, *, /, =, etc..) as well as functions such as read(), write(), print().

! Polymorphism takes this concept one step further and allows user defined methods to be overloaded for any
object type.
Dynamic or Late Binding (Overriding)

Traditional procedural languages support static or early binding, which means that the compiler resolves function calls and references during the compile and link phases.

Object oriented languages provide the advantages of dynamic or late binding which allows the messages to be bound to appropriate methods at run-time. This is implemented by means of checking the number and type of the arguments (message parameters) that are sent to the method.

The same function may be defined up and down the class hierarchy, and at each level the new definition will override the old ones.