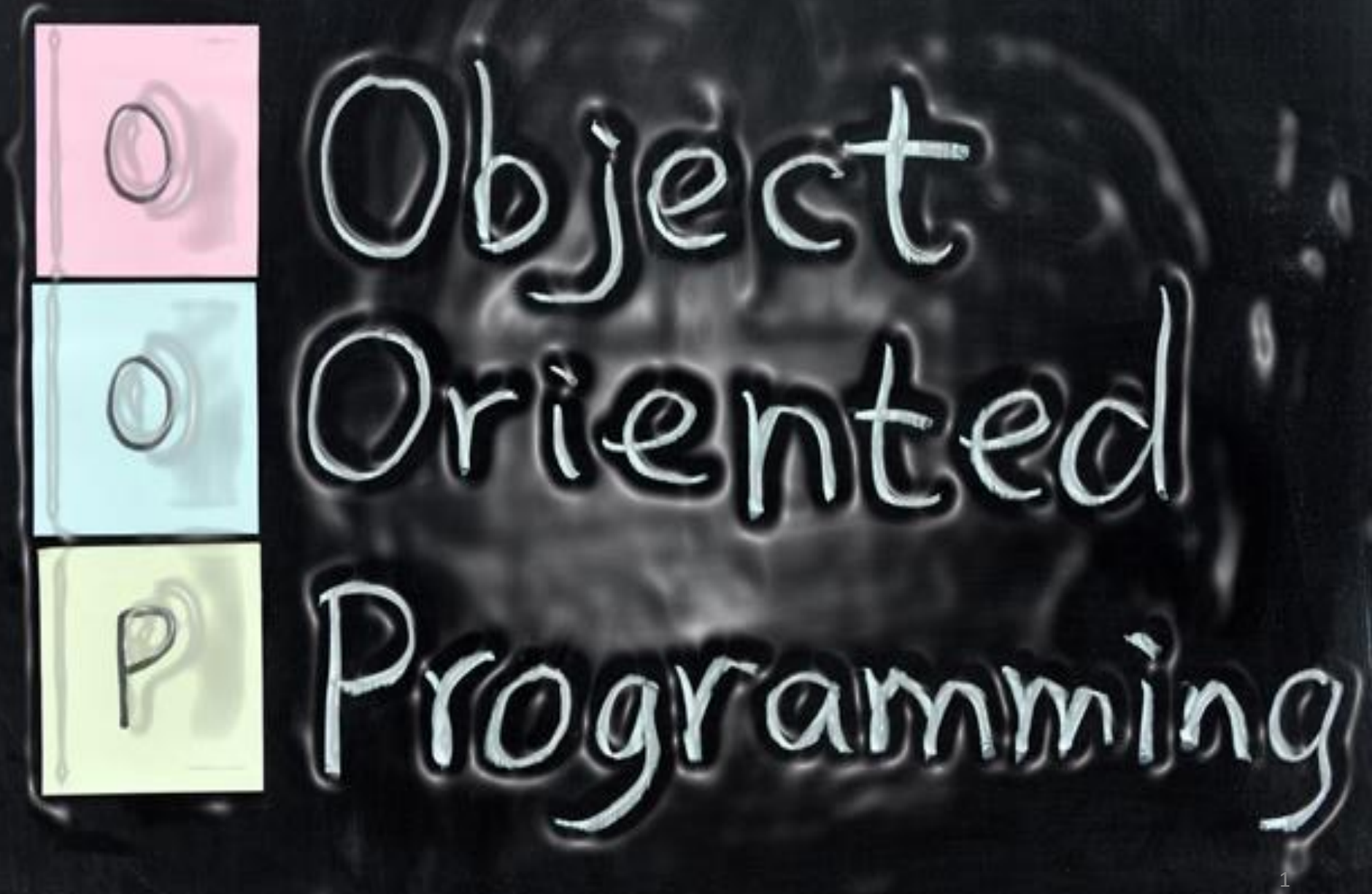
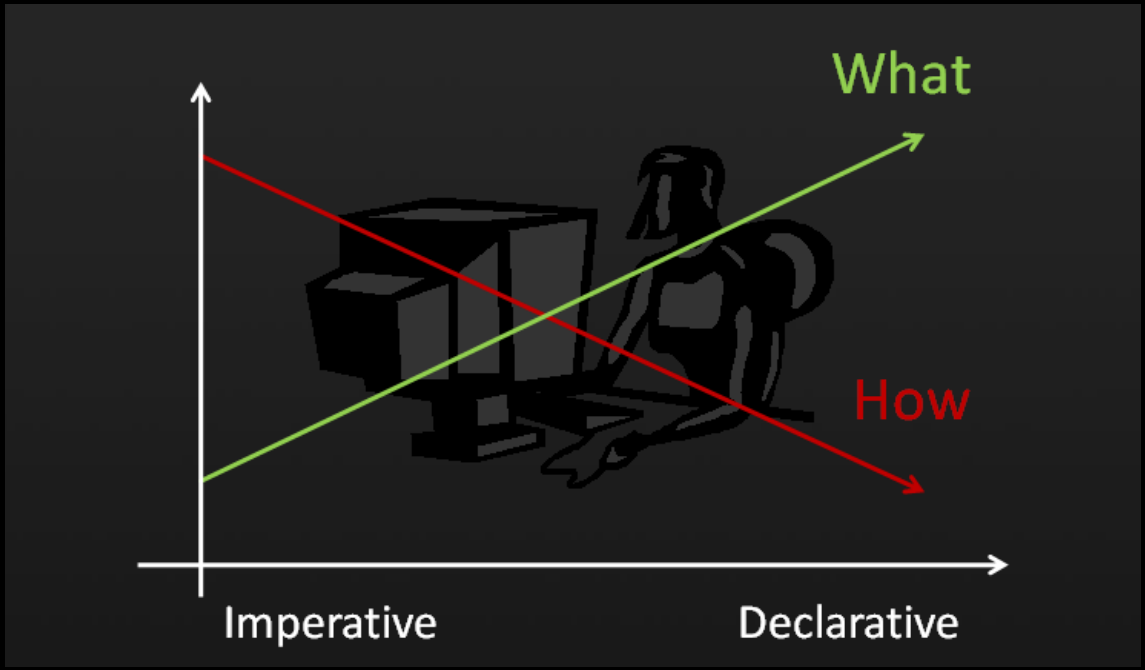

O
O
P
Object
Oriented
Programming

# Objectives

- Review the paradigms of programming languages.
- Introduce the language of Object-Oriented programming.
- Provide explicit details for VB implementation of Object-Oriented programming.

# Computing Language Paradigms

# Computing Language Paradigms




There are two fundamental **programming paradigms**:

- **Declarative programming** centers on **what** computation should be performed
- **Imperative programming centers on how** to compute the problem *explicitly*

# Computing Language Paradigms

> ***Declarative programming*** *is a programming paradigm … that expresses the logic of a computation without describing its control flow.*
>
> ***Imperative programming*** *is a programming paradigm that uses statements that change a program's state.*

# Computing Language Paradigms

> ***Declarative programming*** *is a programming paradigm … that expresses the logic of a computation without describing its control flow.*
>
> ***Imperative programming*** *is a programming paradigm that uses statements that change a program's state.*

- *Declarative Programming* is like asking your friend to draw a landscape. *You don't care how they draw it,* that's up to them.
- *Imperative Programming* is like your friend listening to Bob Ross tell them how to paint a landscape. While good ole Bob Ross isn't exactly commanding, he is giving them *step by step directions* to get the desired result.

# Computing Language Paradigms

> ***Declarative programming*** *is a programming paradigm … that expresses the logic of a computation without describing its control flow.*
>
> ***Imperative programming*** *is a programming paradigm that uses statements that change a program's state.*

- *Declarative Programming* is like asking your friend to draw a landscape. *You don't care how they draw it,* that's up to them.

WHAT TO DO

- *Imperative Programming* is like your friend listening to Bob Ross tell them how to paint a landscape. While good ole Bob Ross isn't exactly commanding, he is giving them *step by step directions* to get the desired result.

HOW TO DO IT

# Computing Language Paradigms

## Functional Paradigm

- We think in terms of **functions** acting on **data**
  - ABSTRACTION: Think of the problem in terms of a process that solves it.
  - DECOMPOSITION: Break your processing down into smaller manageable processing units (functions).
  - ORGANIZATION: Set up your functions so that they call each other (function calls, arguments, etc.)

- FIRST: define your set of data structures (types, etc.)
- THEN: define your set of functions acting upon the data structures.



http://hep.fi.infn.it/JAVAmain.pdf

# Computing Language Paradigms

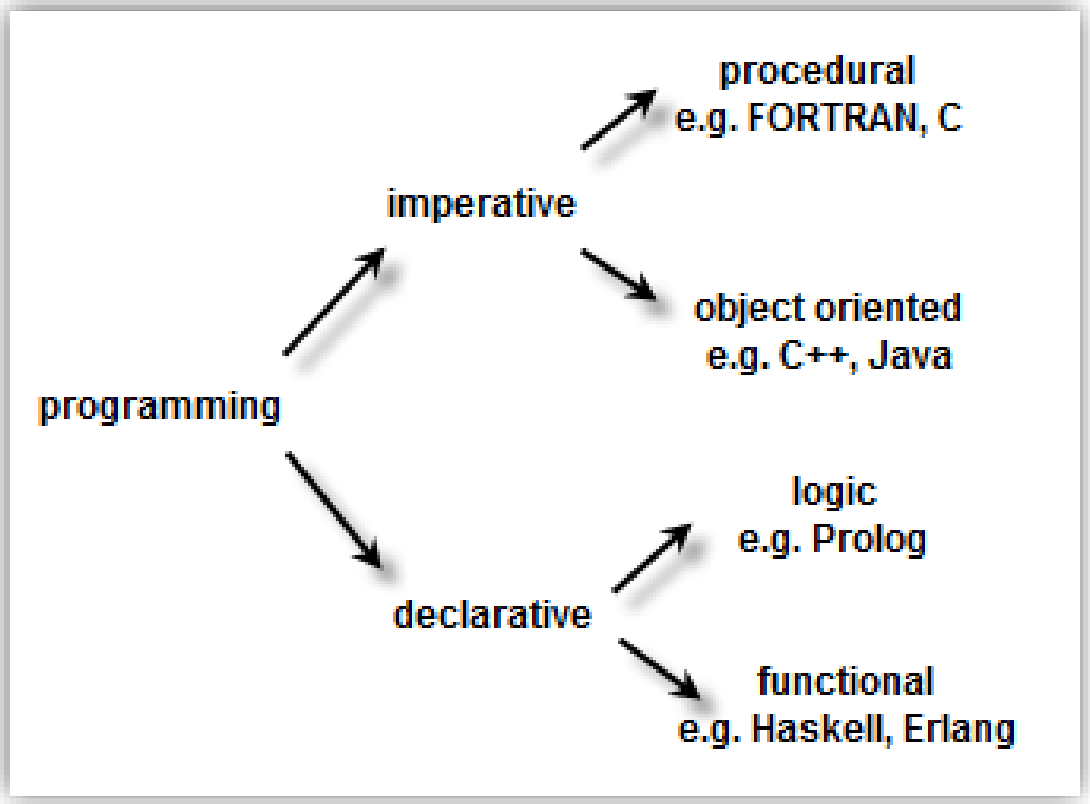
## Object Oriented Paradigm

- We think in terms of objects interacting:
  - ABSTRACTION: Think in terms of independent agents (objects) working together.
  - DECOMPOSITION: Define the kinds of objects on which to split the global task.
  - ORGANIZATION: Create the appropriate number of objects of each kind.
- FIRST: Define the behavior and properties of objects of the different kinds we have defined.
- THEN: Set up objects of each kind and put them to work.

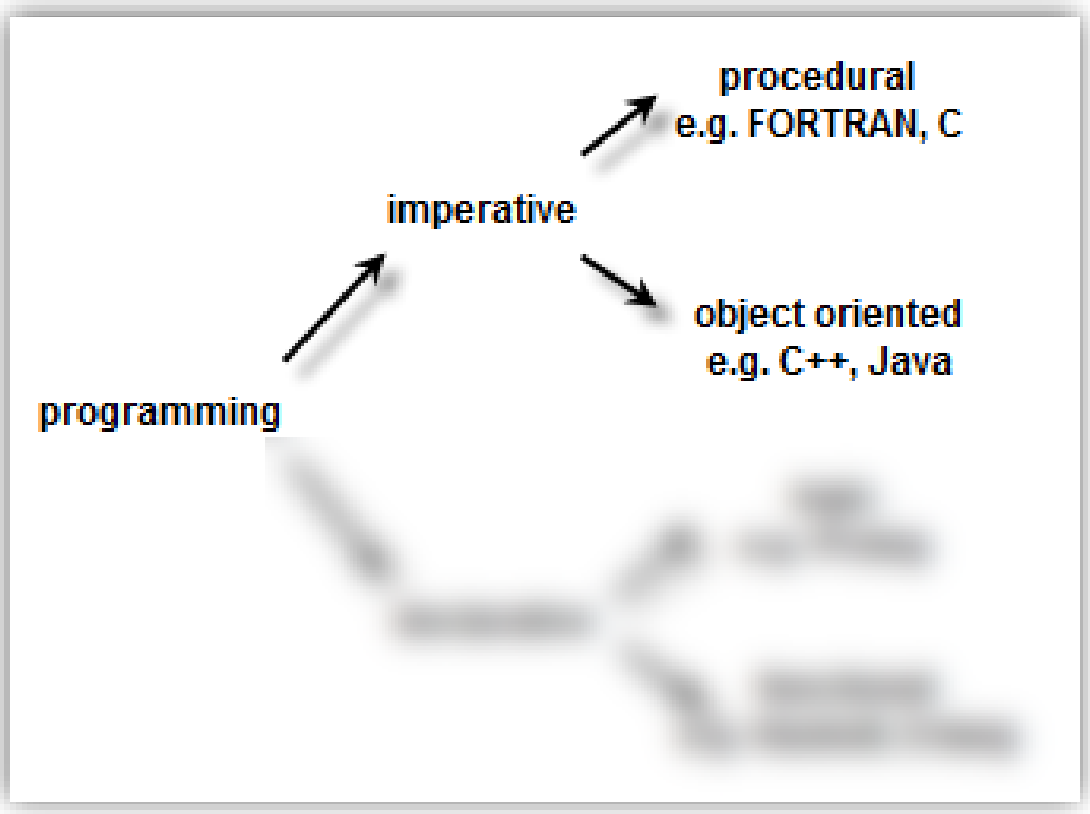


http://hep.fi.infn.it/JAVAmain.pdf

# Computing Language Paradigms



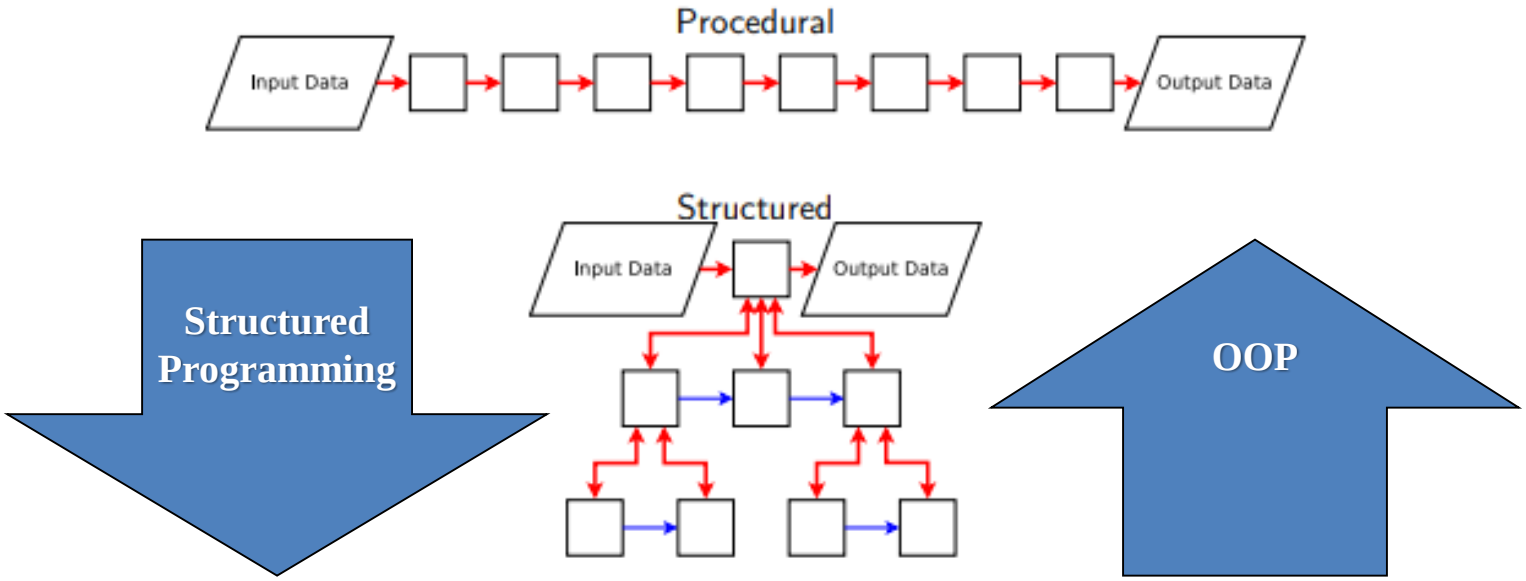
# Detailed View of Imperative Programming




**Top-down design**

**Bottom-up design**

# The OO Paradigm Versus the Structured Paradigm






https://wiki.scinet.utoronto.ca/wiki/images/4/49/Scientific-c%2B%2B.pdf

# OO Abstraction

- In old style programming (procedural), you had:
  - data, which was completely passive, and
  - functions, which could manipulate any data.
- "Object Oriented" is actually a further abstraction along the Imperative Paradigm branch – abstracting the idea of objects beyond simple procedural thinking.
- An object contains both data and methods that manipulate that data
  - An object is *active,* not passive; it *does* things
  - An object is *responsible* for its own data
    - But: it can *expose* that data to other objects

# Abstraction

- Abstraction hides unnecessary details for some immediate purpose.

# Abstraction

- Abstraction hides unnecessary details for some immediate purpose.
- Declarative programming simply *abstracts* away the details of *what* to do.

# Abstraction

- Abstraction hides unnecessary details for some immediate purpose
- Declarative programming simply *abstracts* away the details of *what* to do.
- We have NOT been doing declarative programming.

# Abstraction

- Abstraction hides unnecessary details for some immediate purpose
- Declarative programming simply *abstracts* away the details of *what* to do.
- We have NOT been doing declarative programming.
- Similarly, Object-Oriented programming *abstracts away the actions of programming*, allowing the programmer to focus on real-world **objects** (*i.e.,* OO is a further abstraction of procedural programming within the imperative programming paradigm.

# Definition (*Wikipedia*)

- **Object-oriented programming** (**OOP**):
  A programming [sub-]paradigm that represents the concept of "objects" that have data fields (*attributes* that describe the object) and associated procedures known as *methods*.

# Definition (*Wikipedia*)

- **Object-oriented programming** (**OOP**):
  A programming [sub-]paradigm that represents the concept of "objects" that have data fields (*attributes* that describe the object) and associated procedures known as *methods*.
  - Objects, which are usually *instances of classes*, are used to interact with one another to design applications and computer programs.

# Main Idea Underlying OOP

- Programmers code using “blueprints” of data models called ***classes*** which forms the basis of the following key concepts:
  - **Objects** – A unique programming entity that has *methods*, has *attributes* and can react to *events*.
  - **Methods** – Things which an object can do; the “verbs” of objects. In code, it can usually be identified by an “action” word -- *Hide*, *Show.*
- Examples of OOP languages include C++, Visual Basic.NET and Java.

# Example

- The “*hand*” is a class.
- Your body has two objects of the type "*hand*", named "left hand" and "right hand".
- Their main functions are controlled or managed by a set of electrical signals sent through your shoulders (through an interface).
- So the shoulder is an interface that your body uses to interact with your hands.
- The hand is a well-architected class.
- The hand is being reused to create the left hand and the right hand by slightly changing the properties of it.



https://www.codeproject.com/Articles/22769/Introduction-to-Object-Oriented-Programming-Concep

# Introductory Considerations

# Introductory Considerations

- Rather than structure programs as code and data, an object-oriented system integrates the two using the concept of an "object".

# Introductory Considerations

- Rather than structure programs as code and data, an object-oriented system integrates the two using the concept of an "object".
- An object has state (data) and behavior (code).

# Introductory Considerations

- Rather than structure programs as code and data, an object-oriented system integrates the two using the concept of an "object".
- An object has state (data) and behavior (code).
- Objects correspond to things found in the real world.

  For example,
  - a graphics program will have objects such as *circle*, *square*, *menu*.
  - An online shopping system will have objects such as *shopping cart*, *customer*, *product*. The shopping system will support behaviors such as *place order*, *make payment*, and *offer discount*.

# Introductory Considerations

- Rather than structure programs as code and data, an object-oriented system integrates the two using the concept of an "object".
- An object has state (data) and behavior (code).
- Objects correspond to things found in the real world.

  For example,
  - a graphics program will have objects such as *circle*, *square*, *menu*.
  - An online shopping system will have objects such as *shopping cart*, *customer*, *product*. The shopping system will support behaviors such as *place order*, *make payment*, and *offer discount*.
- The objects are designed as class hierarchies.

  For example,
  - with the shopping system there might be high level classes such as *electronics product*, *kitchen product*, and *book*.
  - There may be further refinements for example under *electronic products*: *Personal Computer, Cell phone, Phased plasma rifle in the 40W range*, *etc.*

# Introductory Considerations

- Rather than structure programs as code and data, an object-oriented system integrates the two using the concept of an "object".
- An object has state (data) and behavior (code).
- Objects correspond to things found in the real world.

  For example,
  - a graphics program will have objects such as *circle*, *square*, *menu*.
  - An online shopping system will have objects such as *shopping cart*, *customer*, *product*. The shopping system will support behaviors such as *place order*, *make payment*, and *offer discount*.
- The objects are designed as class hierarchies.

  For example,
  - with the shopping system there might be high level classes such as *electronics product*, *kitchen product*, and *book*.
  - There may be further refinements for example under *electronic products*: *Personal Computer, Cell phone, Phased plasma rifle in the 40W range, etc.*
- **These classes and subclasses correspond to sets and subsets in mathematical logic.**

# Data Modeling

- 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 generalized 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*.



*http://searchsoa.techtarget.com/definition/object-oriented-programming*

# Benefits of OO Programming

The concepts and rules used in object-oriented programming provide these important benefits:

- The concept of a data class makes it possible to define subclasses of data objects that share some or all of the main class characteristics. Called **inheritance**, this property of OOP forces a more thorough data analysis, reduces development time, and ensures more accurate coding.
- Since a class defines only the data it needs to be concerned with, when an instance of that class (an object) is run, the code will not be able to accidentally access other program data. This characteristic of **data hiding (*a.k.a.* encapsulation)** provides greater system security and avoids unintended data corruption.
- The definition of a class is reusable not only by the program for which it is initially created but also by other object-oriented programs (and, for this reason, can be more easily distributed for use in networks).
- The concept of data classes allows a programmer to create any new data type that is not already defined in the language itself.

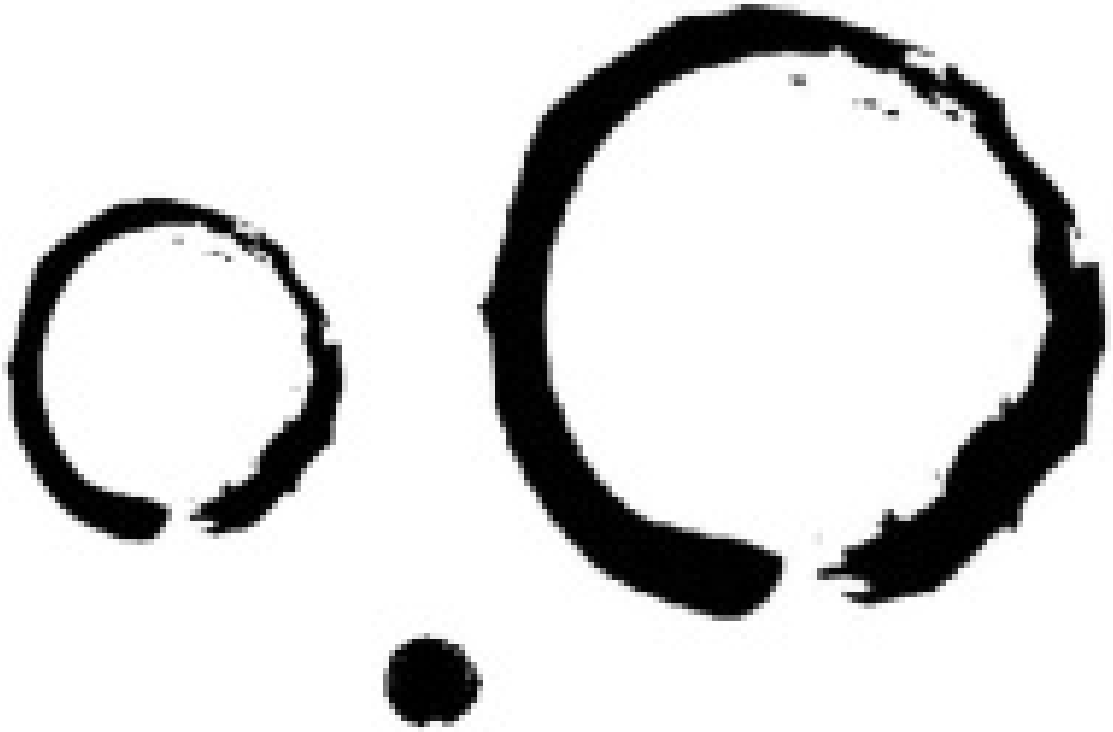


*http://searchsoa.techtarget.com/definition/object-oriented-programming*

# Communication/Synchronization

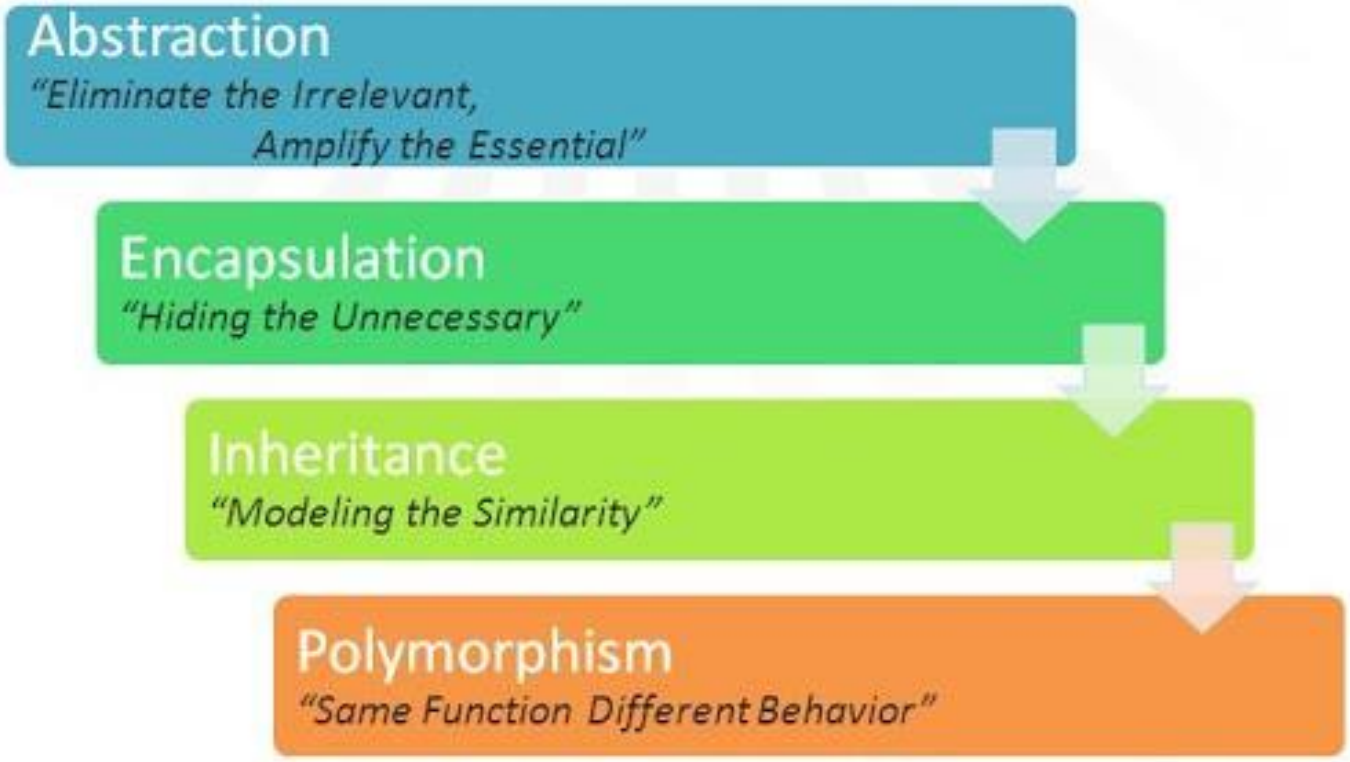
- Sometimes objects need to communicate with one another.
- This is done through Object Oriented Protocols:
  - Definition: OO Protocol
    - In object-oriented programming, a **protocol** or **interface** is a common means for unrelated objects to communicate with each other. These are definitions of methods and values which the objects agree upon in order to cooperate.
    - See http://en.wikipedia.org/wiki/Protocol_%28object-oriented_programming%29

# Key Elements of OO



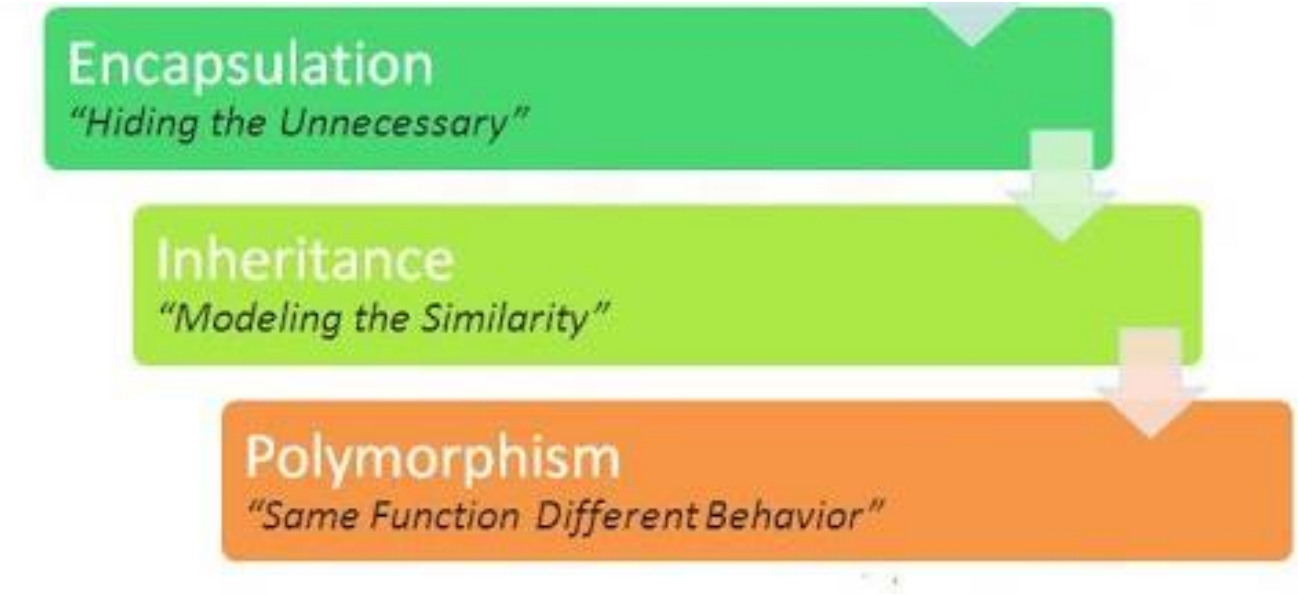
# Key Elements of OO

Object-oriented methodology relies on four main characteristics that define object-oriented languages:



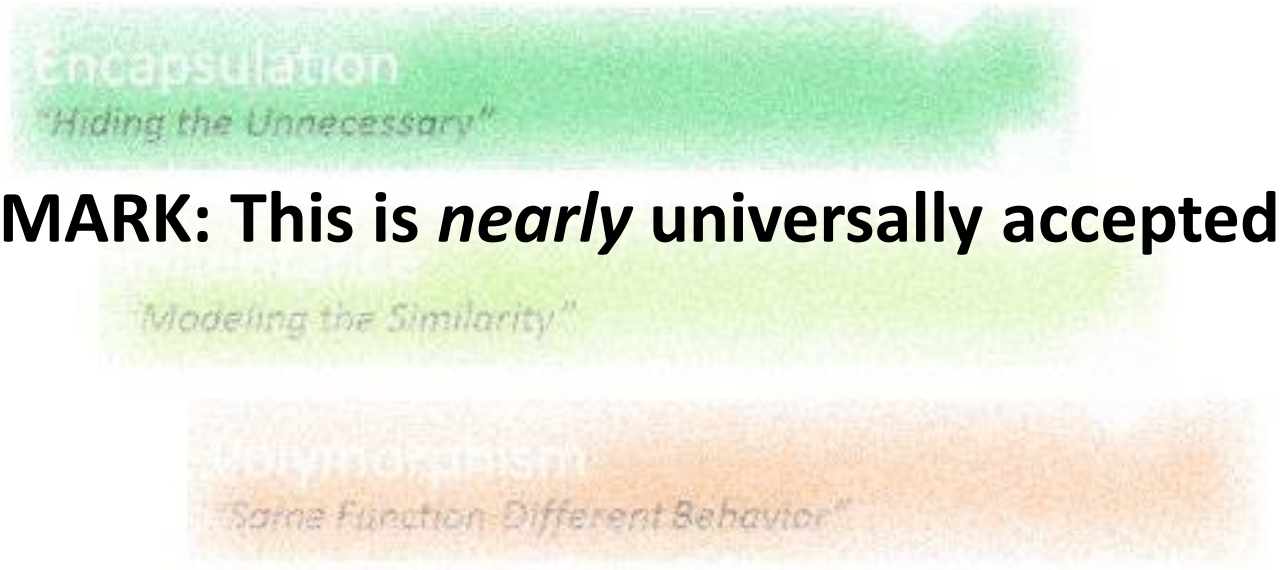
# Key Elements of OO

Abstraction underlies all of these principles so, in effect, there are really only these three characteristics of OO which are most often referred to:



# Key Elements of OO

Object-oriented methodology relies on four main characteristics that define object-oriented languages:




**REMARK: This is *nearly* universally accepted.**

# Key Elements of OO

## Encapsulation

- Hides the behavior of an object from its implementation
- Separates what an object looks like from how it does it implements its behavior.

Nobody but themselves knows how a ***Track*** draws itself or how an ***Event*** calculates its factor



http://hep.fi.infn.it/JAVAmain.pdf

# Key Elements of OO

## Inheritance

- Mechanism by which a class (***subclass***) refines the behavior and properties of some other class (***superclass***).
- The subclass IS A superclass plus something else.

A **ColoredEvent** is an **Event** plus extra data and some redefinitions.

This is **reuse** of code.



http://hep.fi.infn.it/JAVAmain.pdf

# Key Elements of OO

## Polymorphism

- We can deal with objects without the need to know what exact class they belong to
- This is an extension of the inheritance concept

**Sample.some_method** just needs its argument to be an **Event** so it can also be an object of any class derived from **Event.** Actual methods are resolved at run time, by the OO mechisms.



http://hep.fi.infn.it/JAVAmain.pdf

# Example: Simple Car Class

- **properties of the car class:**
  - speed
  - direction
  - engineType
  - frameType
- **methods of the car class:**
  - accelerate()
  - decelerate()
  - turnLeft()
  - turnRight()
- **how the car might be used**
  - DIM myCar as Car
    SET myCar = new Car
    myCar.accelerate()
    msgBox myCar.speedNote that after we create a variable of type car, we can access its properties and methods through the drop-down completion box just as if it had been a built - in class.

  - Note also that we do not deal with the class (Car) directly, but with an INSTANCE of the class (myCar). This is familiar already. We don't deal with the textbox class, but with individual instances of it (txtInput.text).



http://cs.iupui.edu/~aharris/avb/avb4/oop.html

# Example: Simple Car Class Encapsulation

- Code and data should be made as local as possible.
- For example, you shouldn't need to know how every part of the car's engine works.
- If you learn an interface (the steering wheel, accelerator, and displays) you can manage the car without knowing exactly how the fuel injector works.
- Test the code thoroughly, then don't worry about it.
- Design procedures for flexibility, re-use.
- VB encourages encapsulation through modules.
- Our strategy of making all variables as local as possible is also a facet of encapsulation.
- VB encourages encapsulation, but does not require it.
- True OOP lets you think of objects, with all the characteristics and behaviors of the object 'hidden‘, also known as 'data hiding‘.



http://cs.iupui.edu/~aharris/avb/avb4/oop.html

# Example: Simple Car Class Polymorphism

This is a fancy word for a simple idea:

- The same thing can be done in different ways.
  In the physical world, the term 'start' might apply to a car, lawnmower, or motorcycle. In OOP terms, 'start' would be a method of all three objects (remember that methods are actions the object knows how to do). The start method would generally do the same thing in each object, but the underlying behavior would be different. For example, to start a car, you turn a key. To start a motorcycle, you might kick a pedal. To start a lawnmower, you would pull a lanyard. When you look closely at the object, the exact way the object is started is important. However, when you are writing an algorithm that uses the object, you don't really care. We might invoke lawnMower.start(), motorCycle.start(), or car.start(). It would be up to the individual objects to determine how the start method will actually happen.
- Polymorphism is often also used to allow one function to work on multiple parameter types or lists. For example, the msgBox function you are used to takes five parameters. Almost nobody uses all five. We frequently will call this procedure with only one parameter. The ability to still function with different types of parameters is a frequently seen kind of polymorphism.
  VB uses the variant data type to simulate polymorphism, and it supports some other kinds of polymorphism as well.



http://cs.iupui.edu/~aharris/avb/avb4/oop.html

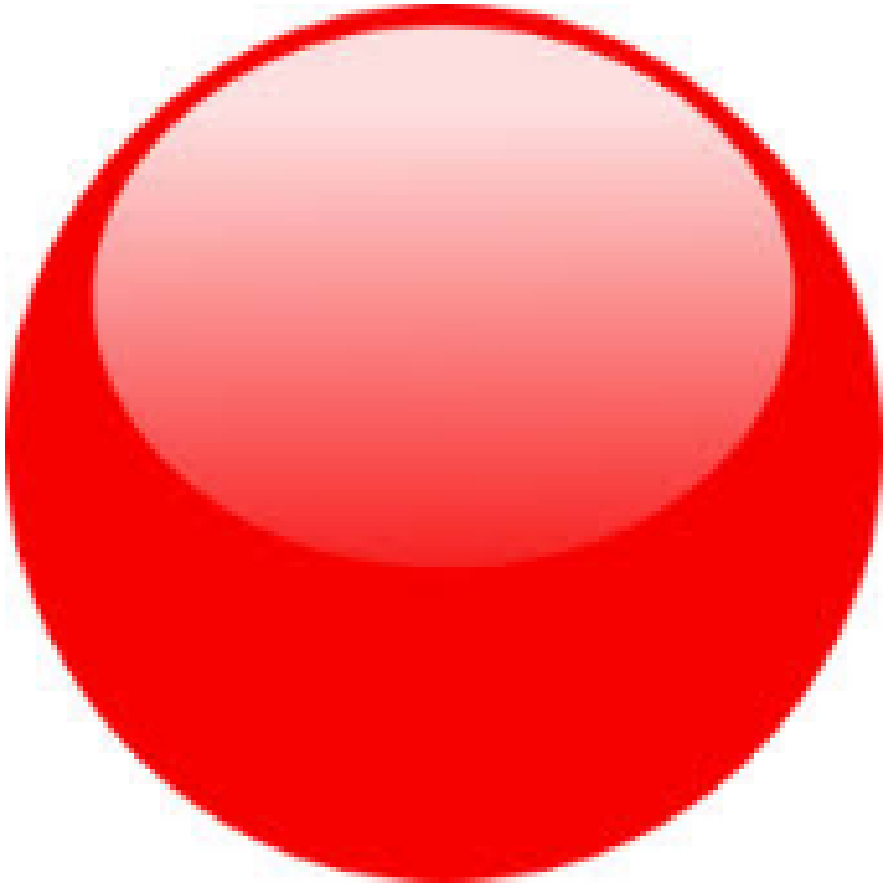
# Example: Simple Car Class Polymorphism

- Inheritance is one of the most important ideas in OOP. Unfortunately, it is the one VB has the most trouble with. The basic idea is quite simple.
- Imagine the car class we discussed above. This would be quite useful on its own, as we could generate new instances of the class to make several different cars. However, what if we wanted to create a police car? It ought to start with the same characteristics as a regular car, but then should add some special features that relate to cop cars, such as a floodLightOn property and a soundSiren() method.
- We could copy and paste all the features of the original car class to a new blank class, then add the new features, but this seems like a waste. It would be much more like the real world if we could take the car class 'straight from the factory' and work on only the modifications needed to make it a police car class. This is what inheritance promises. You can build new classes from existing classes, so you don't have to start from scratch every time.
- Windows shows some evidence of inheritance in its design. If you look closely at the VB controls, they tend to have very similar property and event lists. In particular, the picture box and the form object have strikingly similar lists of properties, events, and methods. This is because they both inherit the same characteristics from a common parent, the windows Rectangle class (which is hidden from us by VB). Have you ever noticed that the command button has a backColor property, but that changing the property has no effect? Why was this property included? Because it was INHERITED from a parent class. The default behavior (of allowing the background color to be changed) was over-ridden by the Microsoft standard that command buttons are always the same color, defined in the user settings.
- VB shows evidence of Window's inheritance features, although it allows very limited support for inheritance in the objects you create within the language.



http://cs.iupui.edu/~aharris/avb/avb4/oop.html

# The Dot Notation of OOP



# The Dot Notation of OOP

Let us apply a particular syntactic standard to what we have learned already regarding OOP:

- Consider the following relationships:
  - Fido is a dog. During a typical day, he does various actions: he eats, runs, sleeps, etc. Here's how an object-oriented programmer might write this:

```
Fido = Dog()
Fido.eats()
Fido.runs()
Fido.sleeps()
```



http://reeborg.ca/docs/oop_py_en/oop.html

# The Dot Notation of OOP

- In addition, Fido has various qualities or attributes.
  - These are variables, like we have seen before except that they "belong" to Fido. He is tall (for a dog) and his hair is black. Here's how the programmer might write the same things:

```
Fido.size = "tall";
Fido.hair_colour = "black";
```



http://reeborg.ca/docs/oop_py_en/oop.html

# The Dot Notation of OOP

- In the object-oriented language, we have the following:

  - `Dog` is an example of a *class* of **objects**.
  - `Fido` is an **instance** (or particular object) in the Dog class.
  - `eats(), runs()` and `sleeps()` are **methods** of the Dog class; 'methods' are essentially like 'functions' which we saw before (the only difference is that they belong in a given class/object/instance).
  - `size` and `hair_colour` are attributes of a given instance/object; attributes can take any value that a "normal" variable can take.
  - The connection between the attributes or the methods with the object is indicated by a "dot" (".") written between them.



http://reeborg.ca/docs/oop_py_en/oop.html

# The Dot Notation of OOP

- Objects can also have other objects that belong to them, each with their own methods or attributes:

```
Fido.tail.wags()
Fido.tail.type = "bushy";
Fido.left_front_paw.moves()
Fido.head.mouth.teeth.canine.hurts()
```



http://reeborg.ca/docs/oop_py_en/oop.html

# The Dot Notation for Classes and Objects in Visual Basic

## Namespaces

A Namespace is a group of Classes which are grouped together. The System.IO Namespace you met earlier groups together Classes that you use to read and write to a file. System.Windows.Forms is another Namespace you've met. In fact, you couldn't create your forms without this Namespace. But again, it is just a group of Classes huddling under the same umbrella.

System itself is a Namespace. It's a top-level Namespace. Think of it as the leader of a hierarchy. IO and Windows would be part of this hierarchy, just underneath the leader. Each subsequent group of Classes is subordinate to the one the came before it. For example Forms is a group of Classes available to Windows, just as Windows is a group of Classes available to System. A single form is a Class available to Forms:

System.Windows.Forms.**Form**

The dot notation is used to separate each group of Classes. A Button is also part of the Forms Class:

System.Windows.Forms.**Button**

As too is a Textbox:

System.Windows.Forms.**TextBox**

http://www.homeandlearn.co.uk/NET/nets11p1.html

HOME & LEARN

# VB.Net Details For OO - **FYI**

## Class Definition

A class definition starts with the keyword **Class** followed by the class name; and the class body, ended by the End Class statement. Following is the general form of a class definition:

```
[ <attributelist> ] [ accessmodifier ] [ Shadows ] [ MustInherit | NotInheritable ]
Class name [ ( Of typelist ) ]
    [ Inherits classname ]
    [ Implements interfacenames ]
    [ statements ]
End Class
```

# VB.Net Details For OO - **FYI**

- ***attributelist*** is a list of attributes that apply to the class. Optional.
- ***accessmodifier*** defines the access levels of the class, it has values as - Public, Protected, Friend, Protected Friend and Private. Optional.
- ***Shadows*** indicate that the variable re-declares and hides an identically named element, or set of overloaded elements, in a base class. Optional.
- ***MustInherit*** specifies that the class can be used only as a base class and that you cannot create an object directly from it, i.e., an abstract class. Optional.
- ***NotInheritable*** specifies that the class cannot be used as a base class.
- ***Partial*** indicates a partial definition of the class.
- ***Inherits*** specifies the base class it is inheriting from.
- ***Implements*** specifies the interfaces the class is inheriting from.

# In Summary:
# The Proposed Purpose of OO

## Object-Oriented Programming: Making Complicated Concepts Simple

Object-oriented (OO) programming is a programming paradigm that includes or relies on the concept of *objects*, encapsulated data structures that have properties and functions and which interact with other objects.

Objects in a program frequently represent real-world objects — for example, an ecommerce website application might have `Customer` objects, `Shopping_cart` objects, and `Product` objects. Other objects might be loosely related to real-world equivalents — like a `Payment_processor` or `Login_form`. Many other objects serve application logic and have no direct real-world parallel — objects that manage authentication, templating, request handling, or any of the other myriad features needed for a working application.

# OO Scope

OBJECT-ORIENTED **ANALYSIS**: Examines the requirements of a system or a problem from the perspective of the classes and objects found in the vocabulary of the problem domain

OBJECT-ORIENTED **DESIGN**: Architectures a system as made of objects and classes, specifying their relationships (like inheritance) and interactions.

OBJECT-ORIENTED **PROGRAMMING**: A method of implementation in which programs are organized as cooperative collections of objects, each of which represents an instance of some class, and whose classes are all members of a hierarchy of classes.

# Exercise

# Exercise 41: Learning To Speak Object Oriented

In this exercise I'm going to teach you how to speak "object oriented." What I'll do is give you a small set of words with definitions you need to know. Then I'll give you a set of sentences with holes in them that you'll have to understand. Finally, I'm going to give you a large set of exercises that you have to complete to make these sentences solid in your vocabulary.



*http://learnpythonthehardway.org/book/ex41.html*

# Exercise 41: Learning To Speak Object Oriented

In this exercise I'm going to teach you how to speak "object oriented." What I'll do is give you a small set of words with definitions you need to know. Then I'll give you a set of sentences with holes in them that you'll have to understand. Finally, I'm going to give you a large set of exercises that you have to complete to make these sentences solid in your vocabulary.

## Word Drills

- class : Tell Python to make a new kind of thing.
- object : Two meanings: the most basic kind of thing, and any instance of some thing.
- instance : What you get when you tell Python to create a class.
- def : How you define a function inside a class.
- self : Inside the functions in a class, self is a variable for the instance/object being accessed.
- inheritance : The concept that one class can inherit traits from another class, much like you and your parents.
- composition : The concept that a class can be composed of other classes as parts, much like how a car has wheels.
- attribute : A property classes have that are from composition and are usually variables.
- is-a : A phrase to say that something inherits from another, as in a "salmon" is-a "fish."
- has-a : A phrase to say that something is composed of other things or has a trait, as in "a salmon has-a mouth."

Take some time to make flash cards for those and memorize them. As usual this won't make too much sense until after you're done with this exercise, but you need to know the base words first.



*http://learnpythonthehardway.org/book/ex41.html*

LEARN PYTHON The Hard Way

## Phrase Drills

Next I have a list of Python code snippets on the left, and the English sentences for them:

**class X(Y)**
"Make a class named X that is-a Y."

**class X(object): def __init__(self, J)**
"class X has-a __init__ that takes self and J parameters."

**class X(object): def M(self, J)**
"class X has-a function named M that takes self and J parameters."

**foo = X()**
"Set foo to an instance of class X."

**foo.M(J)**
"From foo get the M function, and call it with parameters self, J."

**foo.K = Q**
"From foo get the K attribute and set it to Q."

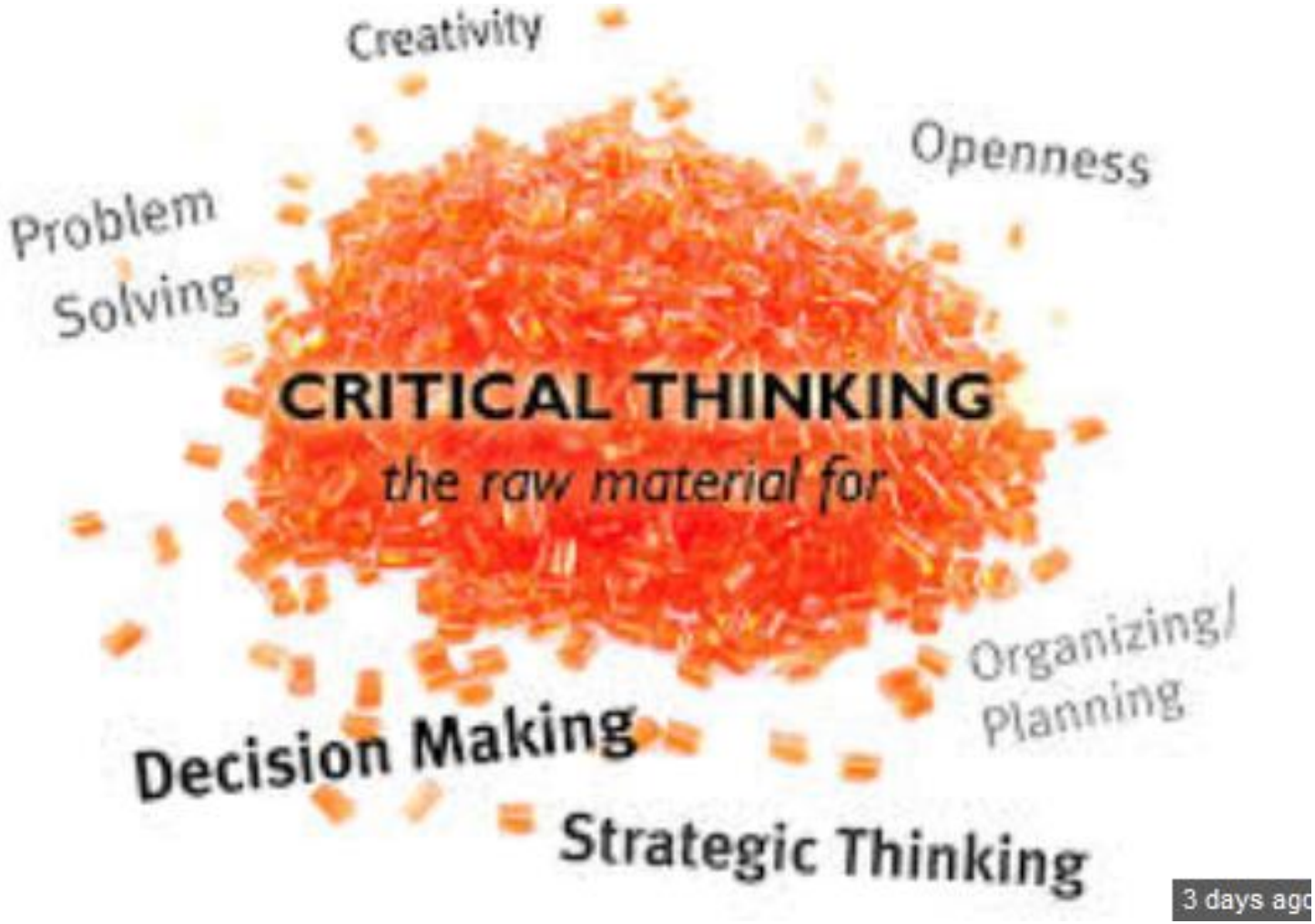
In each of these where you see X, Y, M, J, K, Q, and foo you can treat those like blank spots. For example, I can also write these sentences as follows:

1. "Make a class named ??? that is-a Y."
2. "class ??? has-a __init__ that takes self and ??? parameters."
3. "class ??? has-a function named ??? that takes self and ??? parameters."
4. "Set foo to an instance of class ???."
5. "From foo get the ??? function, and call it with self=??? and parameters ???."
6. "From foo get the ??? attribute and set it to ???."



*http://learnpythonthehardway.org/book/ex41.html*

# What is missing?



# Challenging Assumptions

# Closing Note

The direct **object** is the **noun** that receives the action of the transitive verb. Typically, a direct **object** follows the verb and can be found by asking who or what received the action of the verb. Aug 23, 2013

Nouns: Direct Object - The Tongue Untied
www.grammaruntied.com/blog/?p=671

Programming is based on actions – verbs!
Be critical and don't always go along with the herd:

https://www.youtube.com/watch?v=QM1iUe6IofM

# Closing Note

## Nouns and Verbs

> Java is the most distressing thing to happen to computing since MS-DOS. — Alan Kay

The typical college introduction to OOP starts with a gentle introduction to objects as metaphors for real world concepts. Very few real world OOP programs even consist entirely of nouns, they're filled with **verbs masquerading as nouns: strategies, factories and commands**. Software as a mechanism for directing a computer to do work is primarily concerned with *verbs*.

OOP programs that exhibit low coupling, cohesion and good reusability sometimes feel like nebulous constellations, with hundreds of tiny objects all interacting with each other. Sacrificing readability for changeability.



https://content.pivotal.io/blog/all-evidence-points-to-oop-being-bullshit

# Closing Note

## Inheritance vs. Composition

> Object-oriented programming is an exceptionally bad idea which could only have originated in California — Edsger W. Dijkstra

Inheritance is one of the primary mechanisms for sharing code in an an OO language. But this idea is so problematic that even the keenest advocates of OO **discourage this pattern**. Inheritance forces you to define the taxonomy and structure of your application in advance, with all its connections and intricacies. This structure is **resistant to change** which is one of the primary problems software developers face every day. It also **fails to model** some pretty fundamental concepts.



https://content.pivotal.io/blog/all-evidence-points-to-oop-being-bullshit

# Closing Note

## State

> The problem with object-oriented languages is they've got all this implicit environment that they carry around with them. You wanted a banana but what you got was a gorilla holding the banana and the entire jungle. — Joe Armstrong



https://content.pivotal.io/blog/all-evidence-points-to-oop-being-bullshit

# Presentation Terminated