INTRODUCTION TO OBJECT-ORIENTED PROGRAMMING (OOP)

By

Dr.Pramod S.Nair
Professor & Head
Computer Science and Engineering
Medi-Caps University

SOFTWARE ISSUES

- How to represent real-life problems in system design
- How to ensure reusability and extensibility of modules
- How to develop modules that withstand/ flexible to changes
- How to improve software productivity
- How to decrease software cost
- How to improve the quality of software
- How to manage time schedules

PROGRAMMING LANGUAGES

- Programming languages allow programmers to code software.
- The three major families of languages are:
  - Machine languages
  - Assembly languages
  - High-Level languages

MACHINE LANGUAGES

- Comprised of 1s and 0s
- The "native" language of a computer
- Difficult to program – one misplaced 1 or 0 will cause the program to fail.
- Example of code:

1110100010101 111010101110
10111010110100 10100011110111

The only language understood by a computer is machine language.

ASSEMBLY LANGUAGES

- Assembly languages are a step towards easier programming.
- Assembly languages are comprised of a set of elemental commands which are tied to a specific processor.
- Assembly language code needs to be translated to machine language before the computer processes it.

  Example:

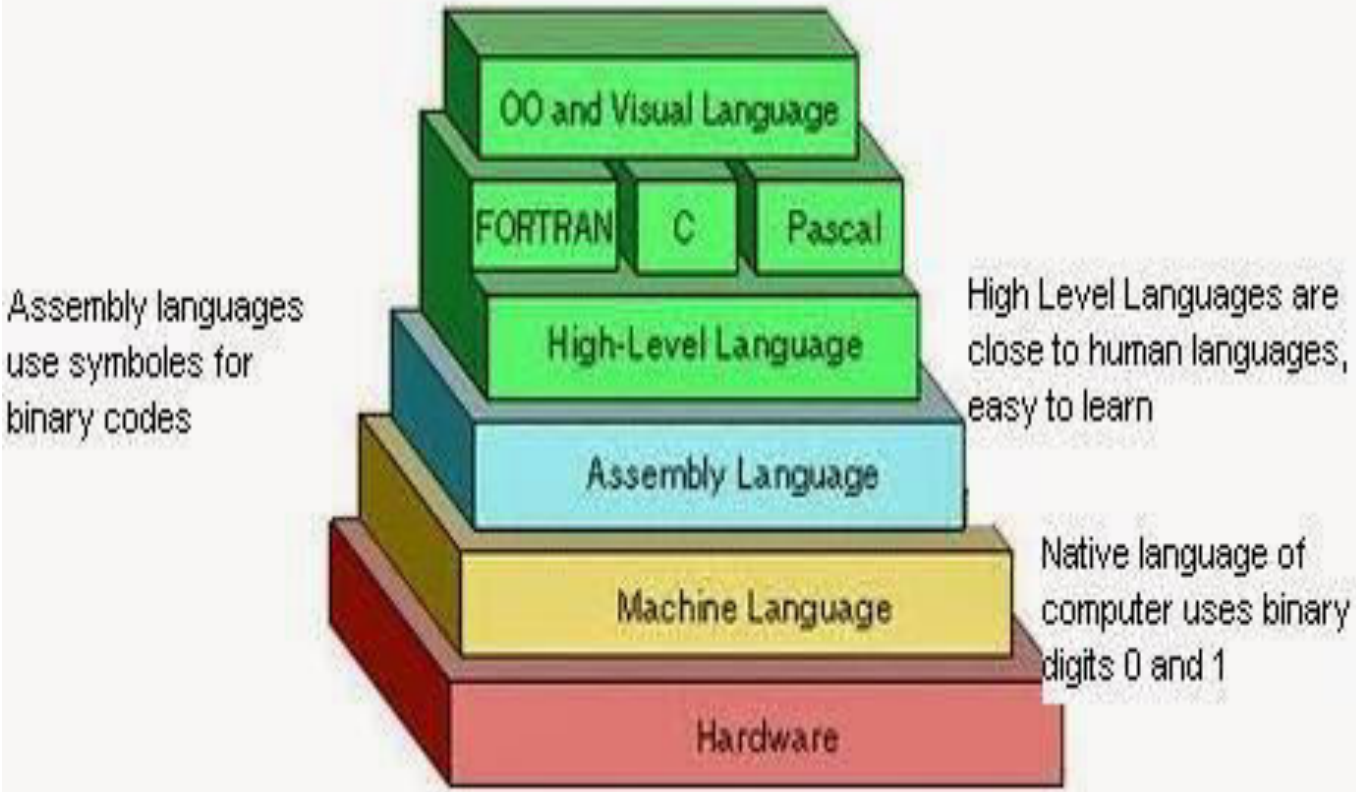
  ADD 1001010, 1011010

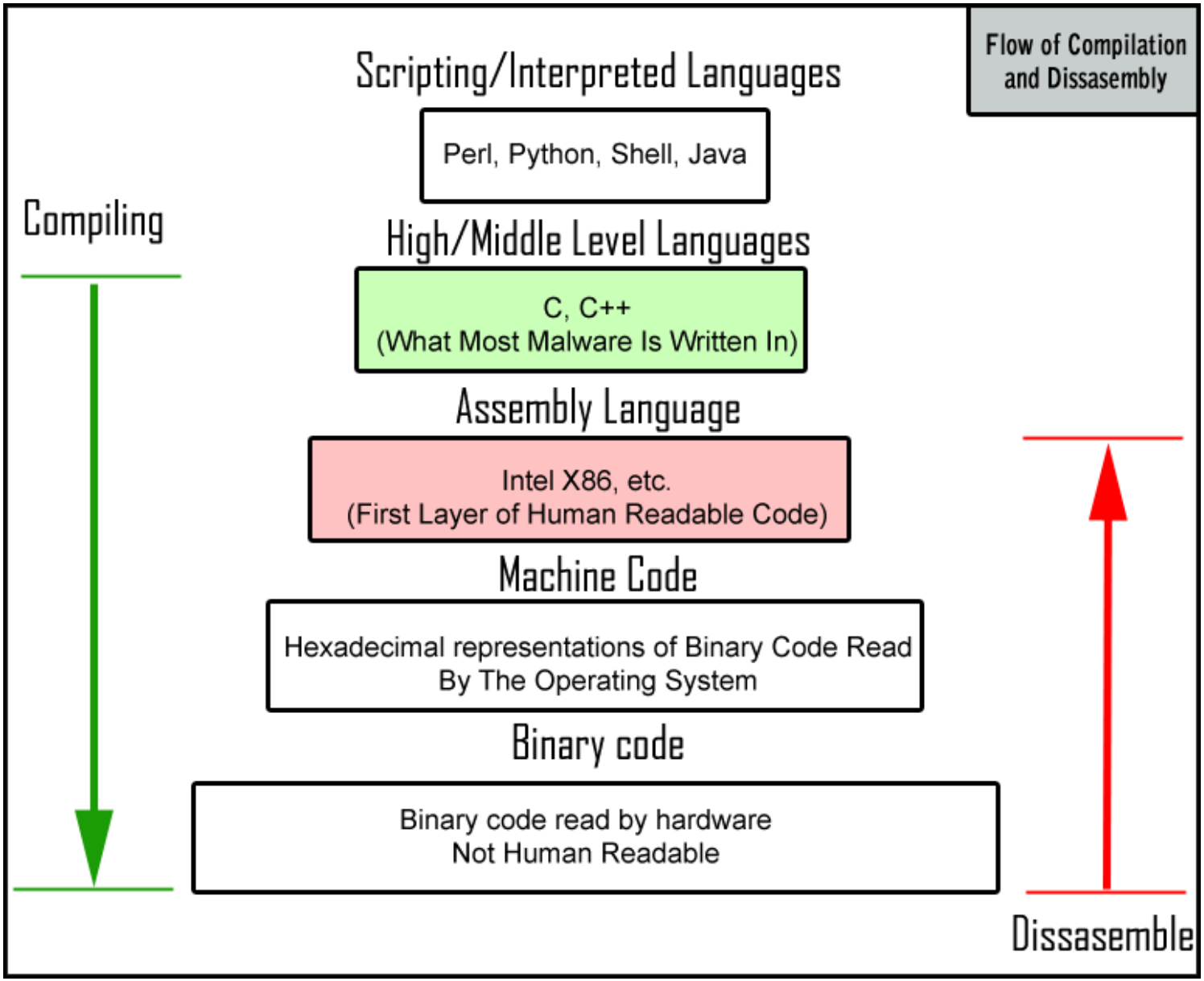
HIGH-LEVEL LANGUAGES

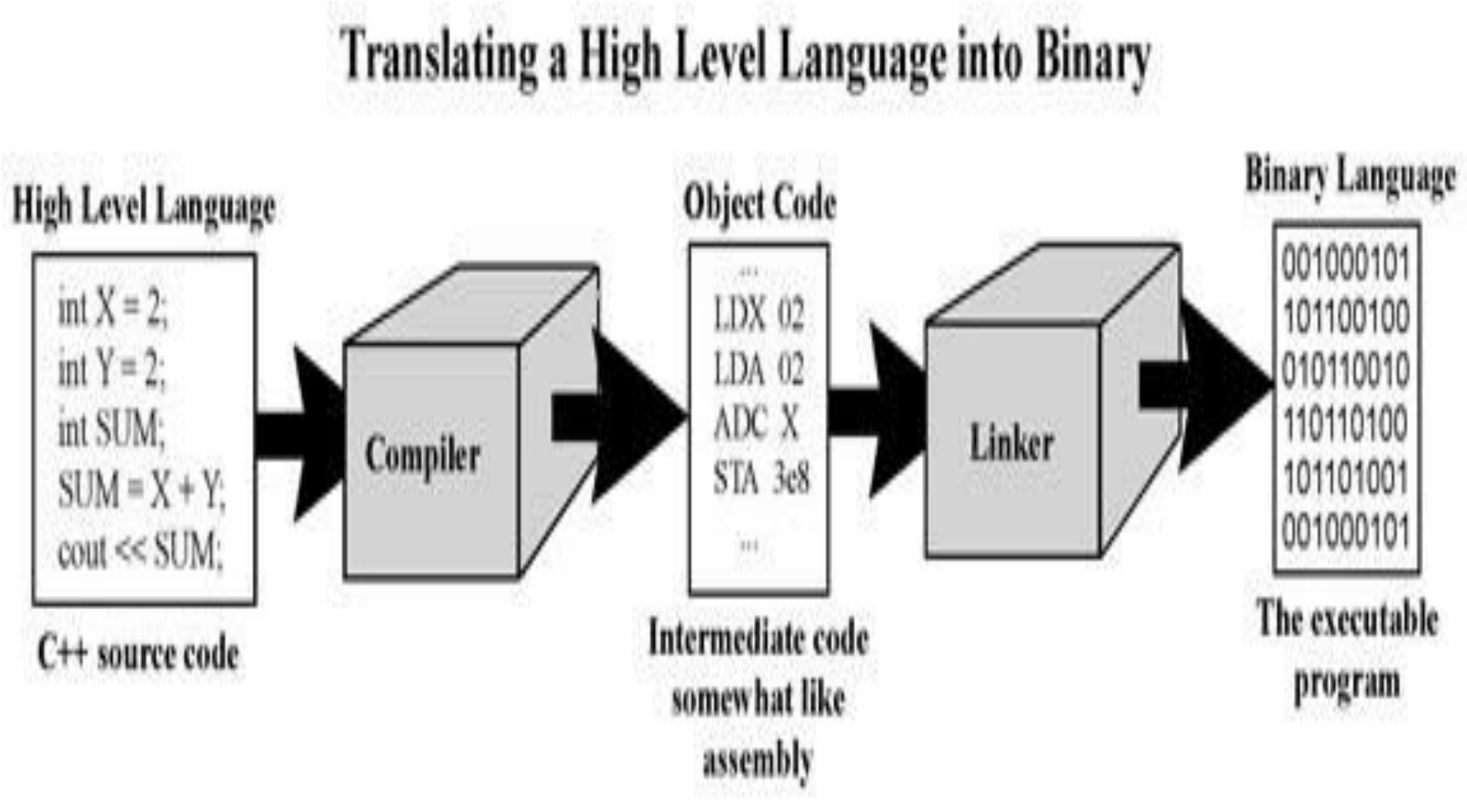
- High-level languages represent a giant leap towards easier programming.
- The syntax of HL languages is similar to English.
- We divide HL languages into two groups:
  - Procedural languages
  - Object-Oriented languages (OOP)

TRANSLATION

- Programs are normally written in one of the high-level languages.
- To run the program on a computer, the program needs to be translated into the machine language of the computer on which it will run.
- The program in a high-level language is called the source program.
- The translated program in machine language is called the object program.
- Two methods are used for translation: compilation and interpretation.


Assembly languages use symboles for binary codes
OO and Visual Language
FORTRAN
C
Pascal
High-Level Language
Assembly Language
Machine Language
Hardware
High Level Languages are close to human languages, easy to learn
Native language of computer uses binary digits 0 and 1

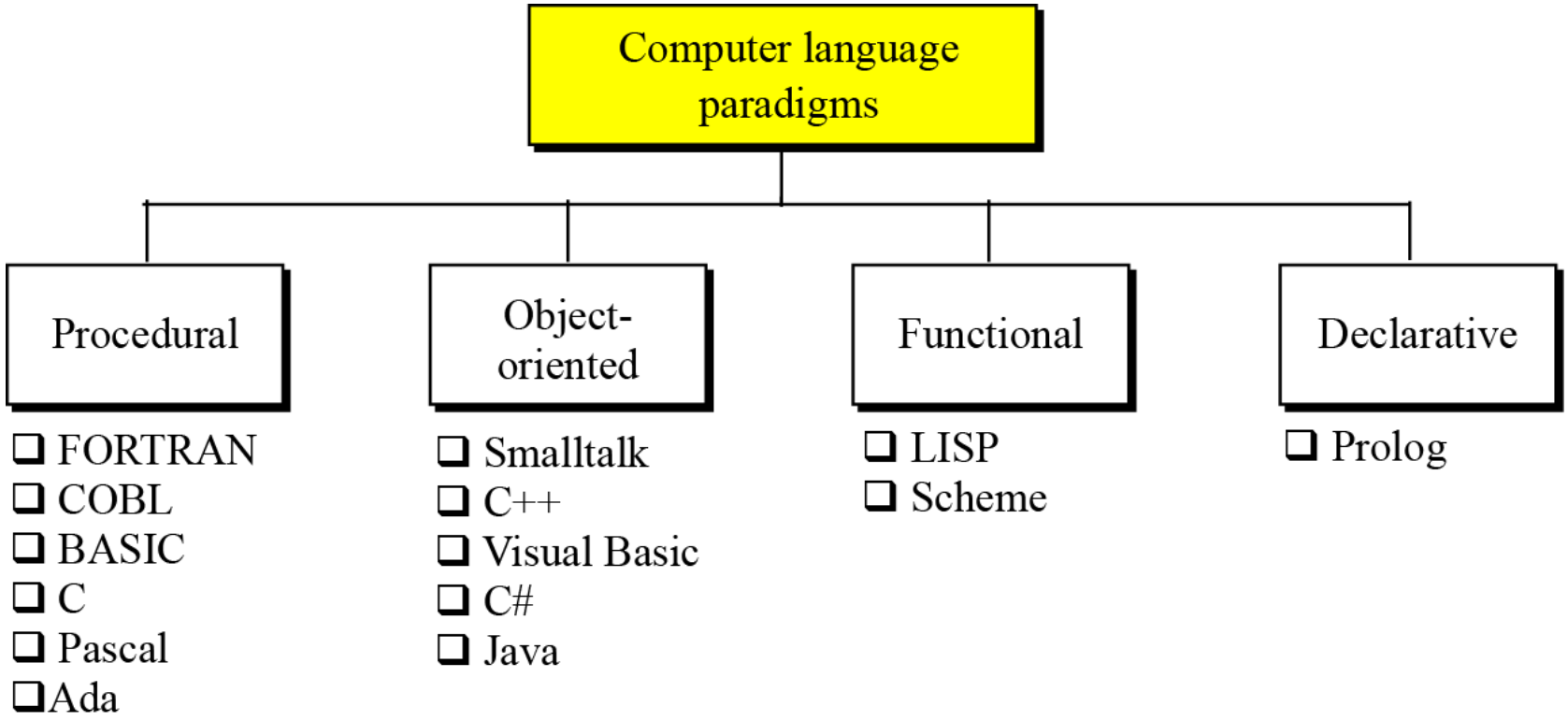

Flow of Compilation and Dissasembly
Scripting/Interpreted Languages
Perl, Python, Shell, Java
Compiling
High/Middle Level Languages
C, C++
(What Most Malware Is Written In)
Assembly Language
Intel X86, etc.
(First Layer of Human Readable Code)
Machine Code
Hexadecimal representations of Binary Code Read By The Operating System
Binary code
Binary code read by hardware
Not Human Readable
Dissasemble



Linker

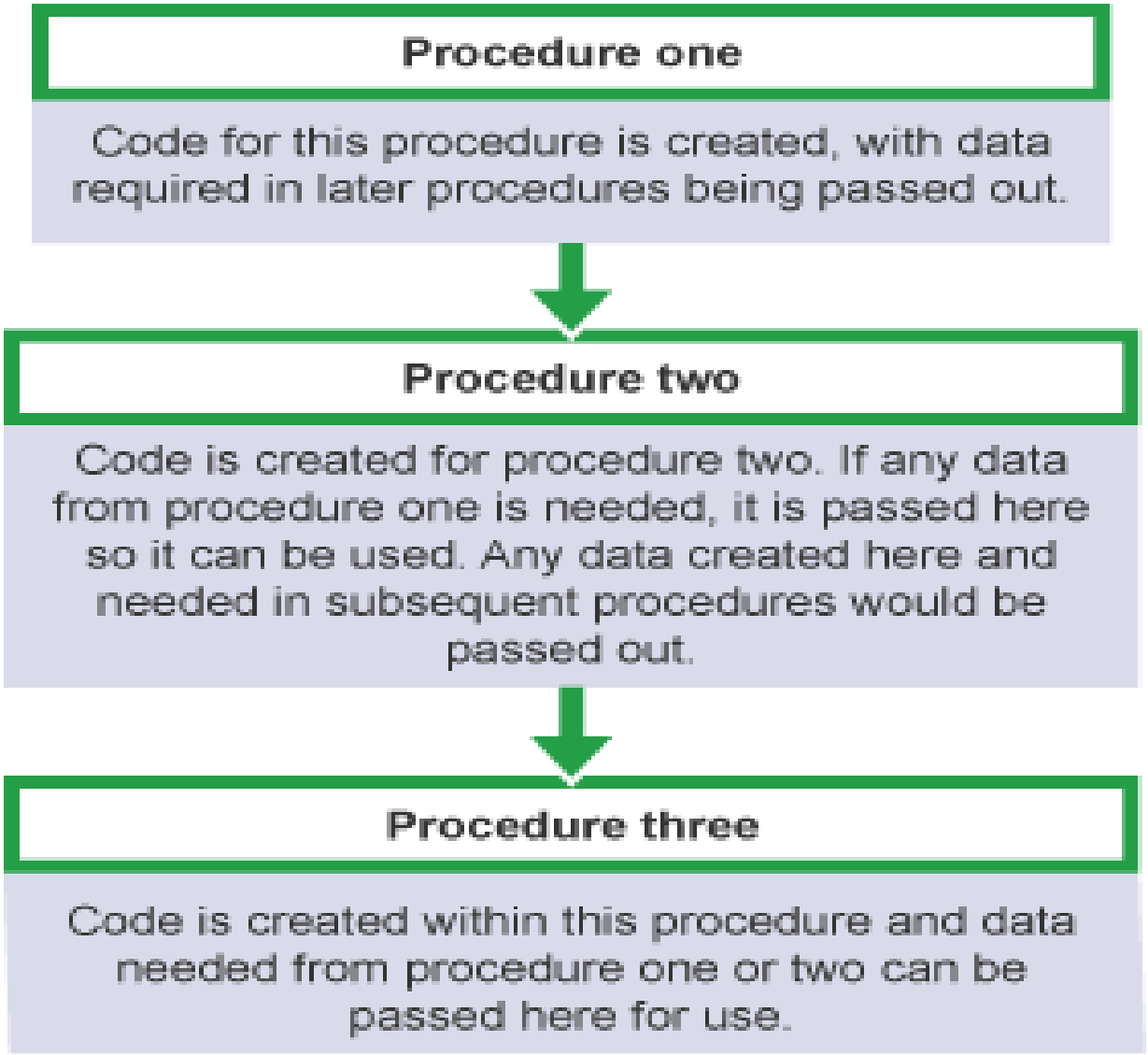
A program used with compiler or assembler to provide links to the libraries needed for an executable program.

CATEGORIES OF PROGRAMMING LANGUAGES



PROCEDURAL LANGUAGES

- Early high-level languages are typically called procedural languages.
- Procedural languages are characterized by sequential sets of linear commands. The focus of such languages is on *structure*.
- Examples include C, COBOL, Fortran, LISP, Perl, HTML, VBScript


Procedure one
Code for this procedure is created, with data required in later procedures being passed out.
Procedure two
Code is created for procedure two. If any data from procedure one is needed, it is passed here so it can be used. Any data created here and needed in subsequent procedures would be passed out.
Procedure three
Code is created within this procedure and data needed from procedure one or two can be passed here for use.

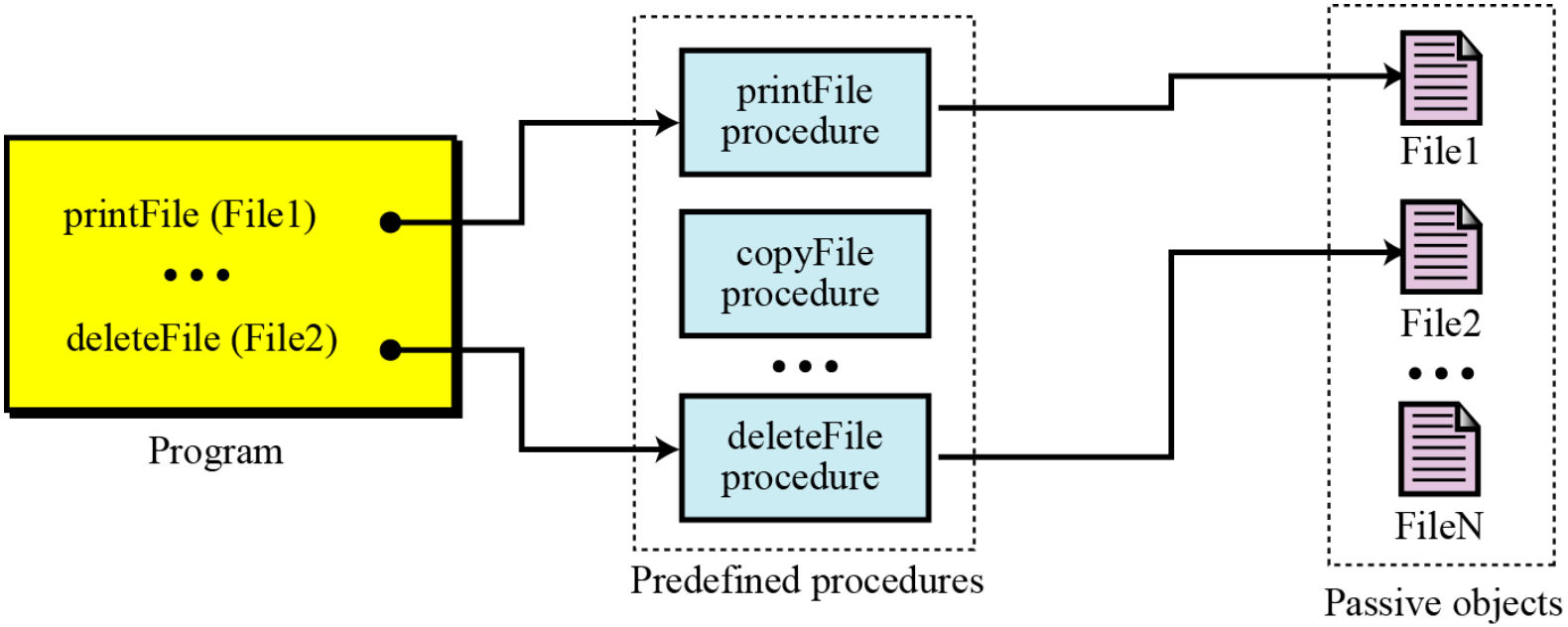
THE PROCEDURAL PARADIGM

- In the procedural paradigm (or imperative paradigm) we can think of a program as an active agent that manipulates passive objects.
- We encounter many passive objects in our daily life: a stone, a book, a lamp, and so on.
- A passive object cannot initiate an action by itself, but it can receive actions from active agents.
- A program in a procedural paradigm is an active agent that uses passive objects that we refer to as data or data items.

THE PROCEDURAL PARADIGM

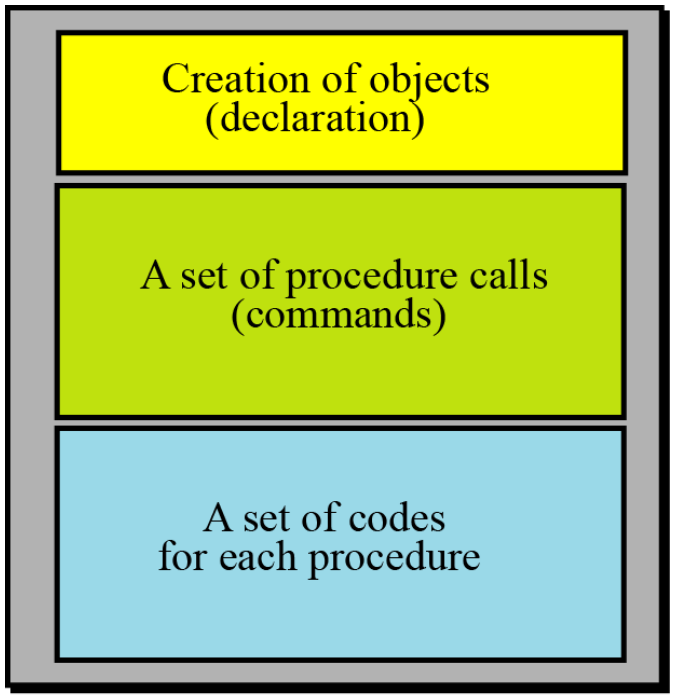
- To manipulate a piece of data, the active agent (program) issues an action, referred to as a procedure.
- For example, think of a program that prints the contents of a file. The file is a passive object.
- To print the file, the program uses a procedure, which we call print.

THE CONCEPT OF THE PROCEDURAL PARADIGM



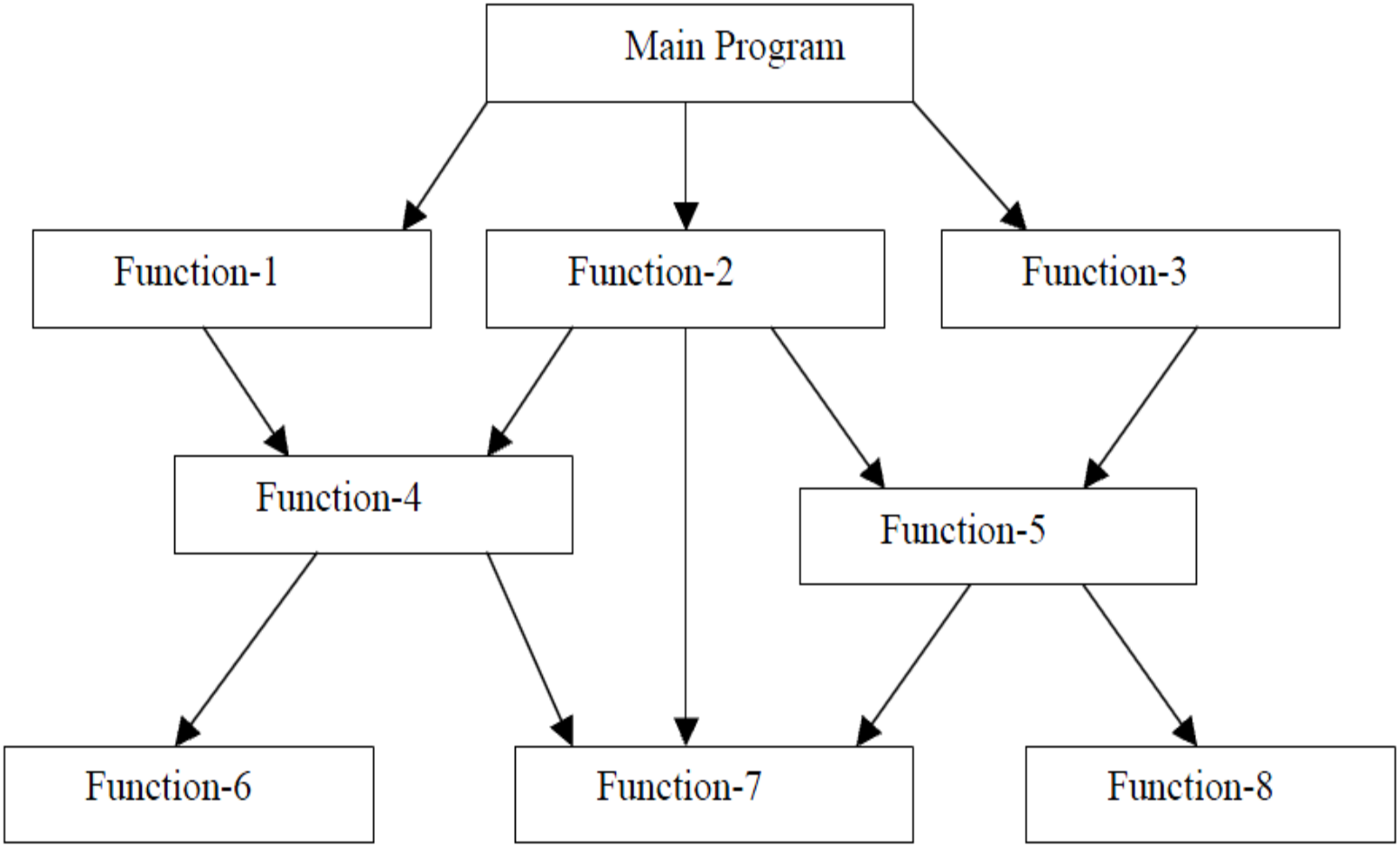
THE COMPONENTS OF A PROCEDURAL PROGRAM

A program in this paradigm is made up of three parts: *a part for object creation*, *a set of procedure calls* and *a set of code for each procedure*. Some procedures have already been defined in the language itself. By combining this code, the programmer can create new procedures.



A procedural program

PROCEDURE-ORIENTED PROGRAMMING



Typical structure of procedural oriented programs

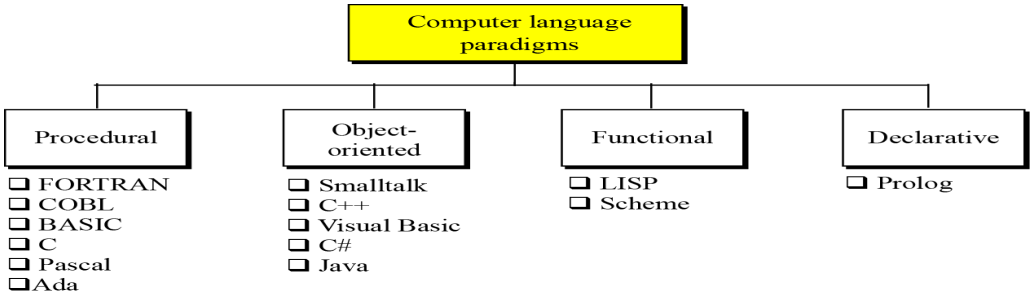
PROCEDURE-ORIENTED PROGRAMMING ISSUES

- Global and local Variables
- If we revise the external data structure then we have to revise all the functions that access the data
- Difficult to model real world problems because functions are action oriented

PROCEDURE-ORIENTED PROGRAMS

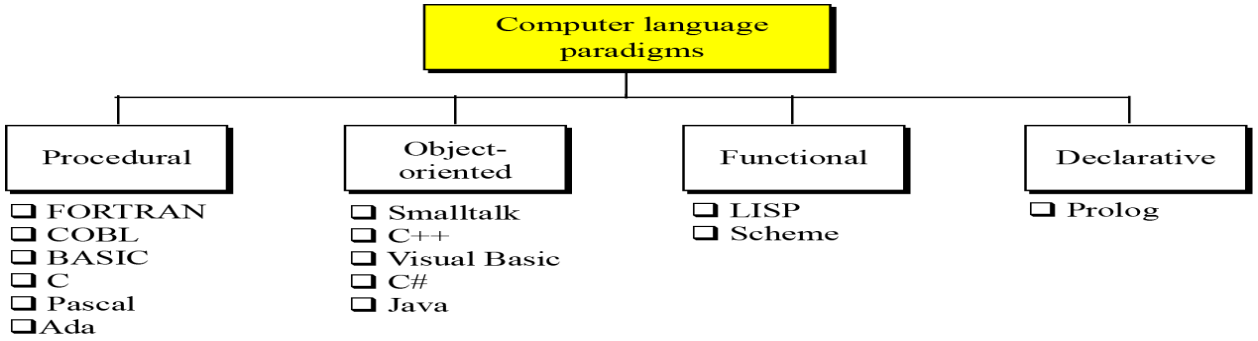
- Emphasis on algorithms
- Problems are divided in to small functions
- Open/free data movement from function to function
- Functions transform data from one form to another
- Top-down approach in program design

DECLARATIVE PROGRAMMING



- **Declarative** programming is when you write your code in such a way that it describes what you want to do, and not how you want to do it. It is left up to the compiler to figure out the how.

- Examples of **declarative** programming **languages** are SQL and Prolog.



LISP

What is LISP?

- **A LISt Processing language**
  - The basic data structure is linked list and Atoms.
- **A functional programming language**
  - Each expression in LISP is a function that returns a value
- **An interpretive language**
  - Running LISP programs involves interacting with the LISP **interpreter**.
  - **Clisp** is the common lisp interpreter available only for **Linux**.
  - Recently Compilers have been made for this language but they are not used a lot.

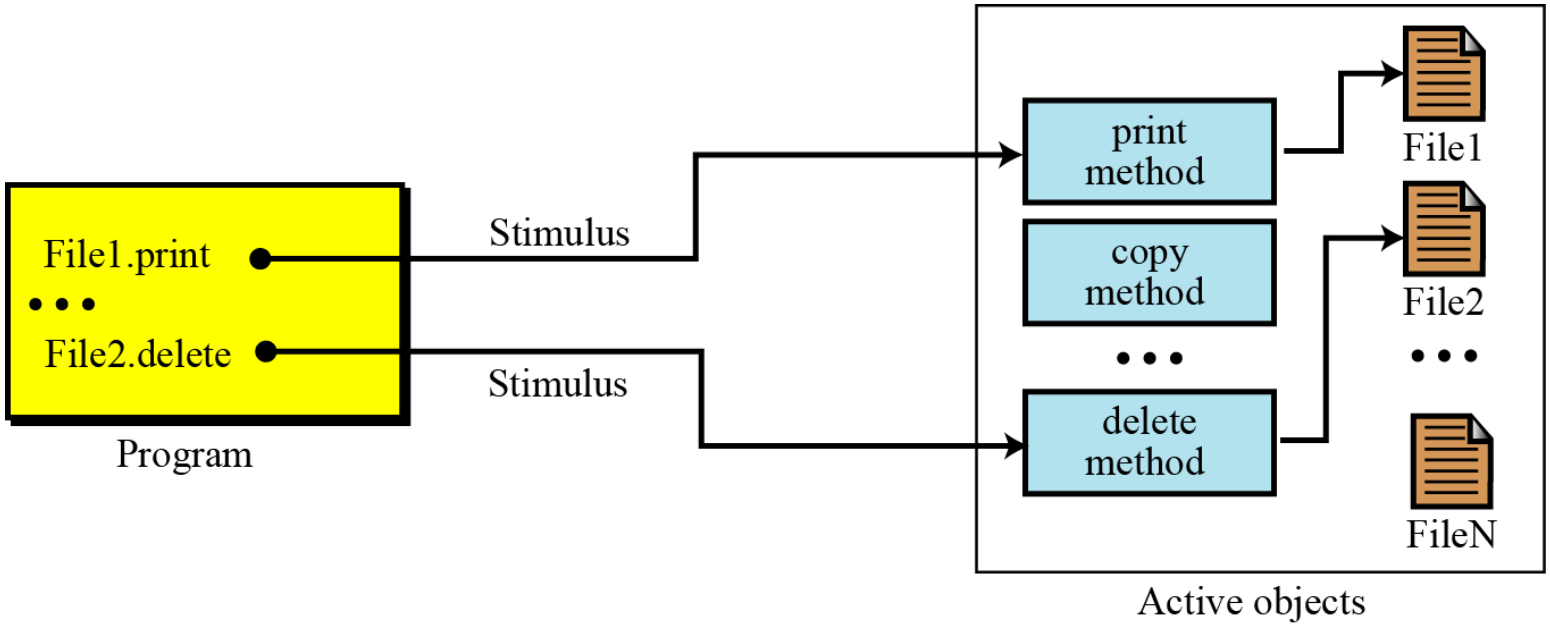
THE OBJECT-ORIENTED PARADIGM

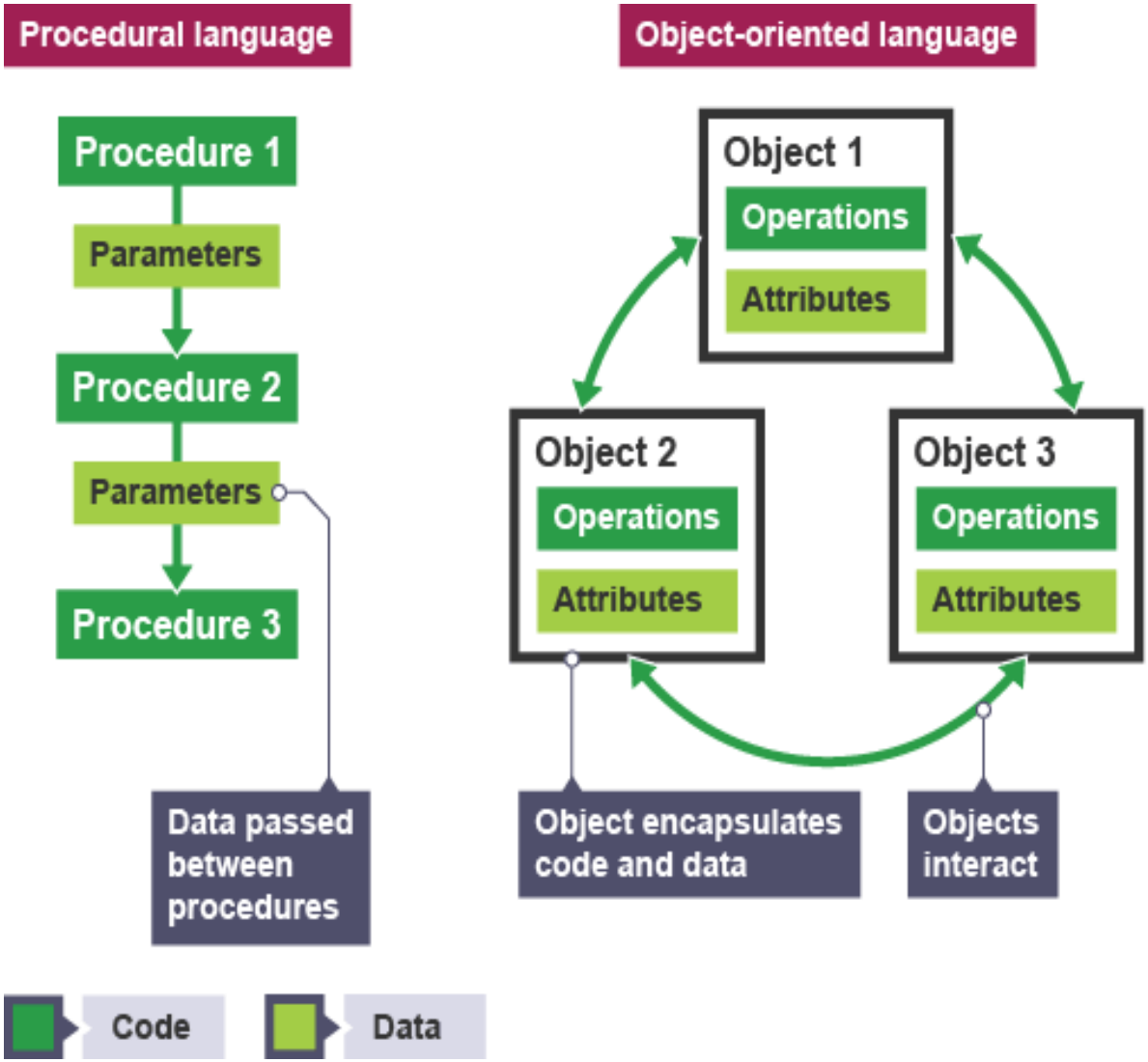
- The object-oriented paradigm deals with active objects instead of passive objects.
- We encounter many active objects in our daily life: a vehicle, an automatic door, a dishwasher and so on.
- The action to be performed on these objects are included in the object: the objects need only to receive the appropriate stimulus from outside to perform one of the actions.

THE OBJECT-ORIENTED PARADIGM

- A file in an object-oriented paradigm can be packed with all the procedures—called methods in the object-oriented paradigm—to be performed by the file: printing, copying, deleting and so on.
- The program in this paradigm just sends the corresponding request to the object.

THE CONCEPT OF AN OBJECT-ORIENTED PARADIGM




Procedural language
Procedure 1
Parameters
Procedure 2
Parameters
Procedure 3
Data passed between procedures
Code
Data
Object-oriented language
Object 1
Operations
Attributes
Object 2
Operations
Attributes
Object 3
Operations
Attributes
Object encapsulates code and data
Objects interact

OBJECT ORIENTED PARADIGM

- Treats data as a critical element in the program development
- It ties data closely to the function that operate on it.
- OOP allows decomposition of a problem into objects
- The data of an object can be accessed through its function
- But function of one object can be accessed by one function of another object

FEATURES OF OOP

- Emphasis on data rather than procedure
- Data structures characterize the objects
- Data is hidden and cannot be accessed by external function
- Objects communicates through functions
- New data and functions can be added whenever necessary
- Follows bottom up approach in program design

BASIC CONCEPTS OF OOP

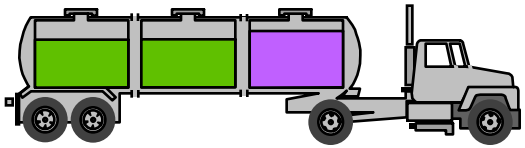
- Objects
- Classes
- Data abstraction and encapsulation
- Inheritance
- Polymorphism
- Dynamic Binding
- Message passing

OBJECTS

- Objects are variable of the type class
- Any number of objects can be defined of the same class
- Class is a collection of objects
- The basic building block of an object-oriented language such as Java– is a template that describes the data and behavior associated with instances of that class.
- Eg: Fruit is a class and the objects are mango, apple etc

WHAT IS AN OBJECT?

- Informally, an object represents an entity, either physical, conceptual, or software
  - Physical entity



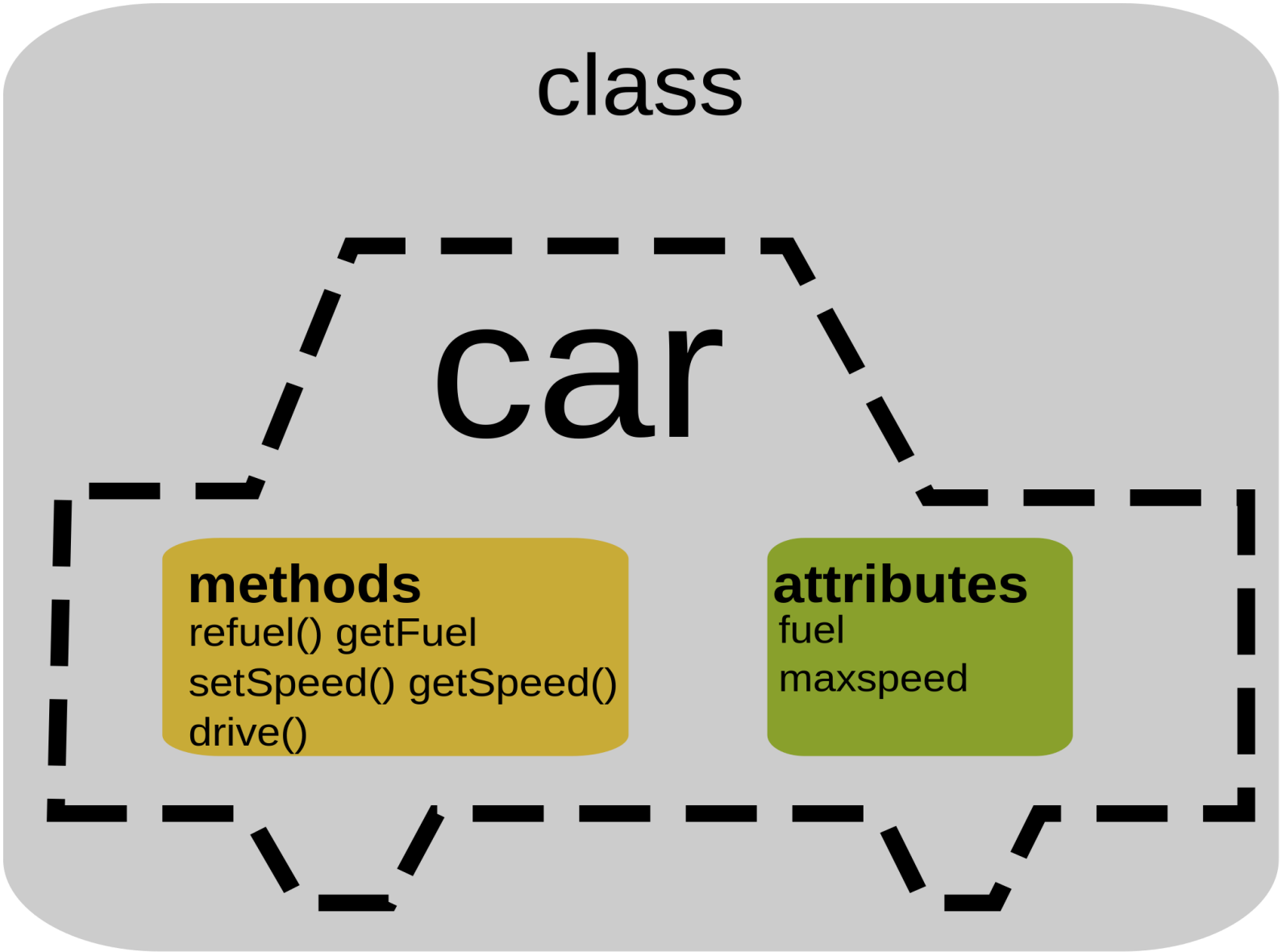
Truck

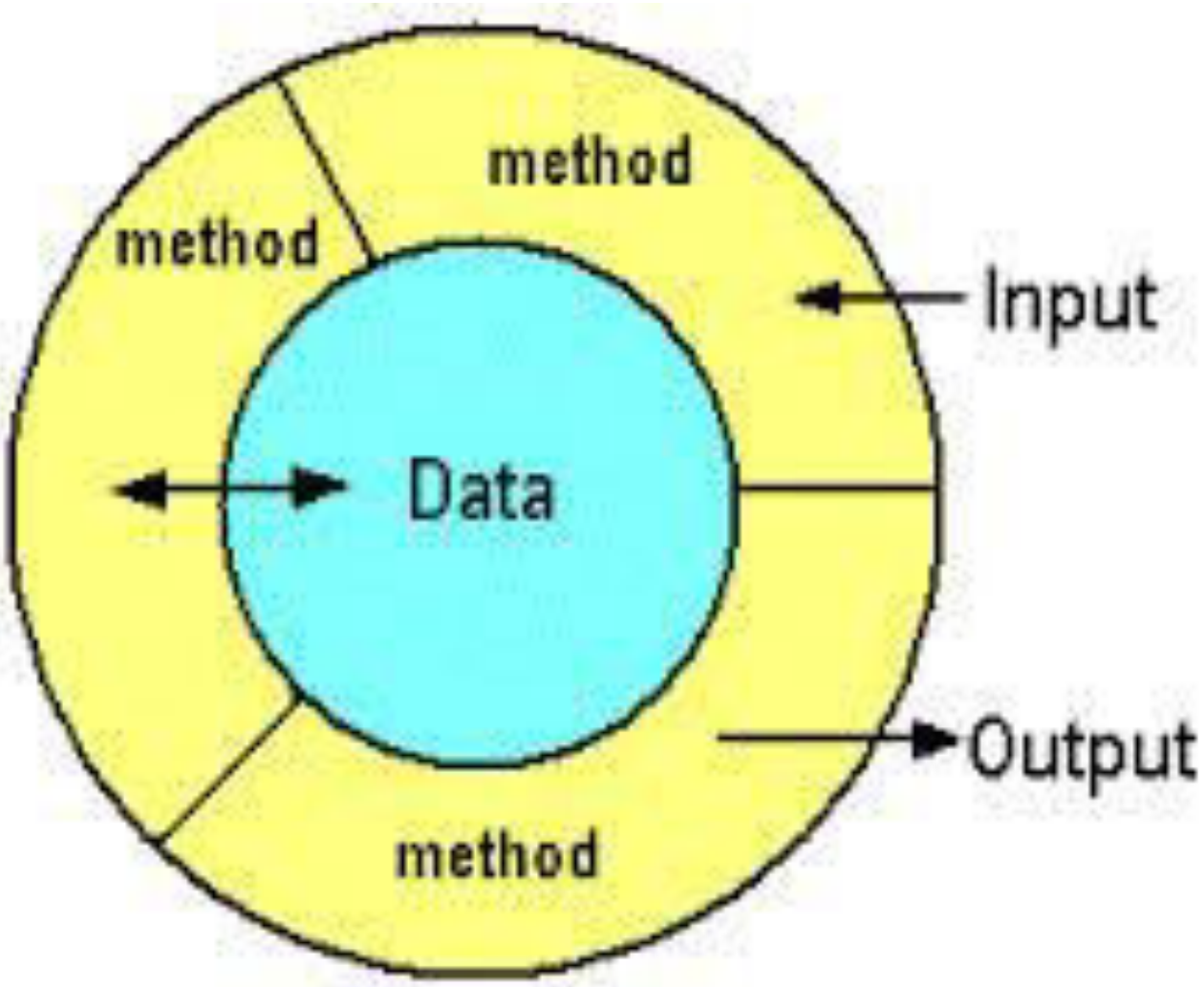
  - Conceptual entity

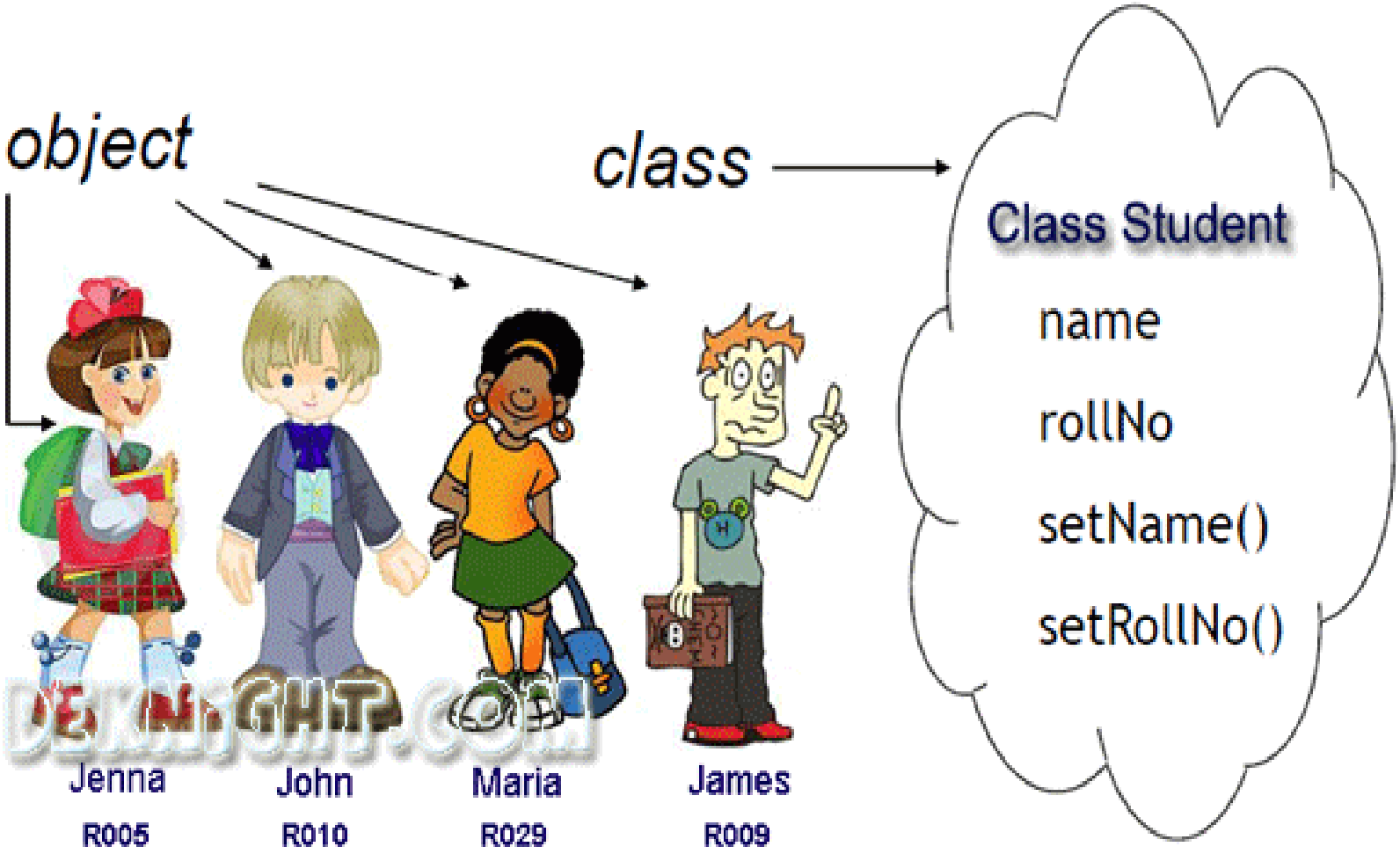
Chemical Process

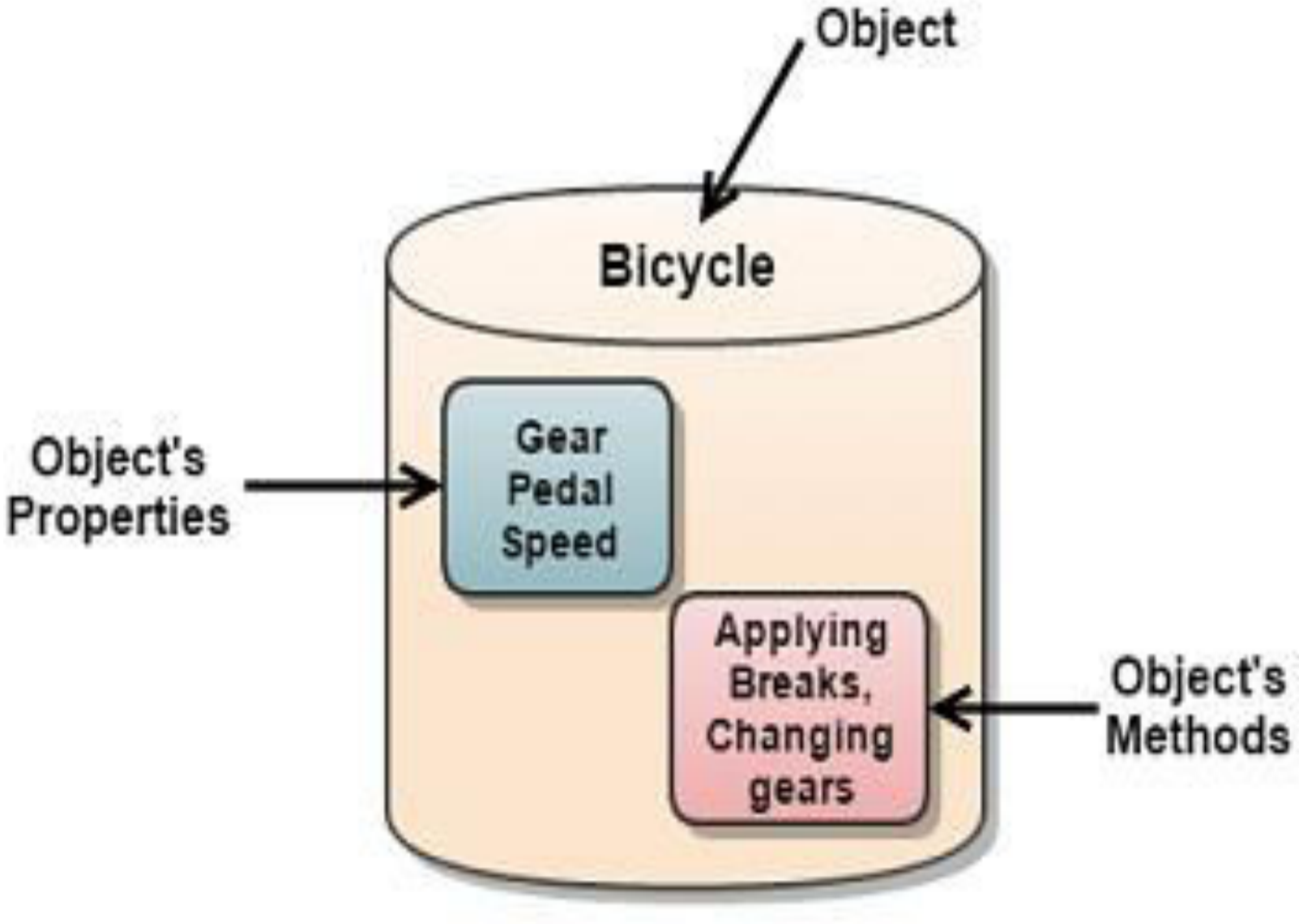
  - Software entity

Linked List


class
car
methods
refuel() getFuel
setSpeed() getSpeed()
drive()
attributes
fuel
maxspeed


method
method
Input
Data
Output
method


object
class
Class Student
name
rollNo
setName()
setRollNo()
Jenna
R005
John
R010
Maria
R029
James
R009

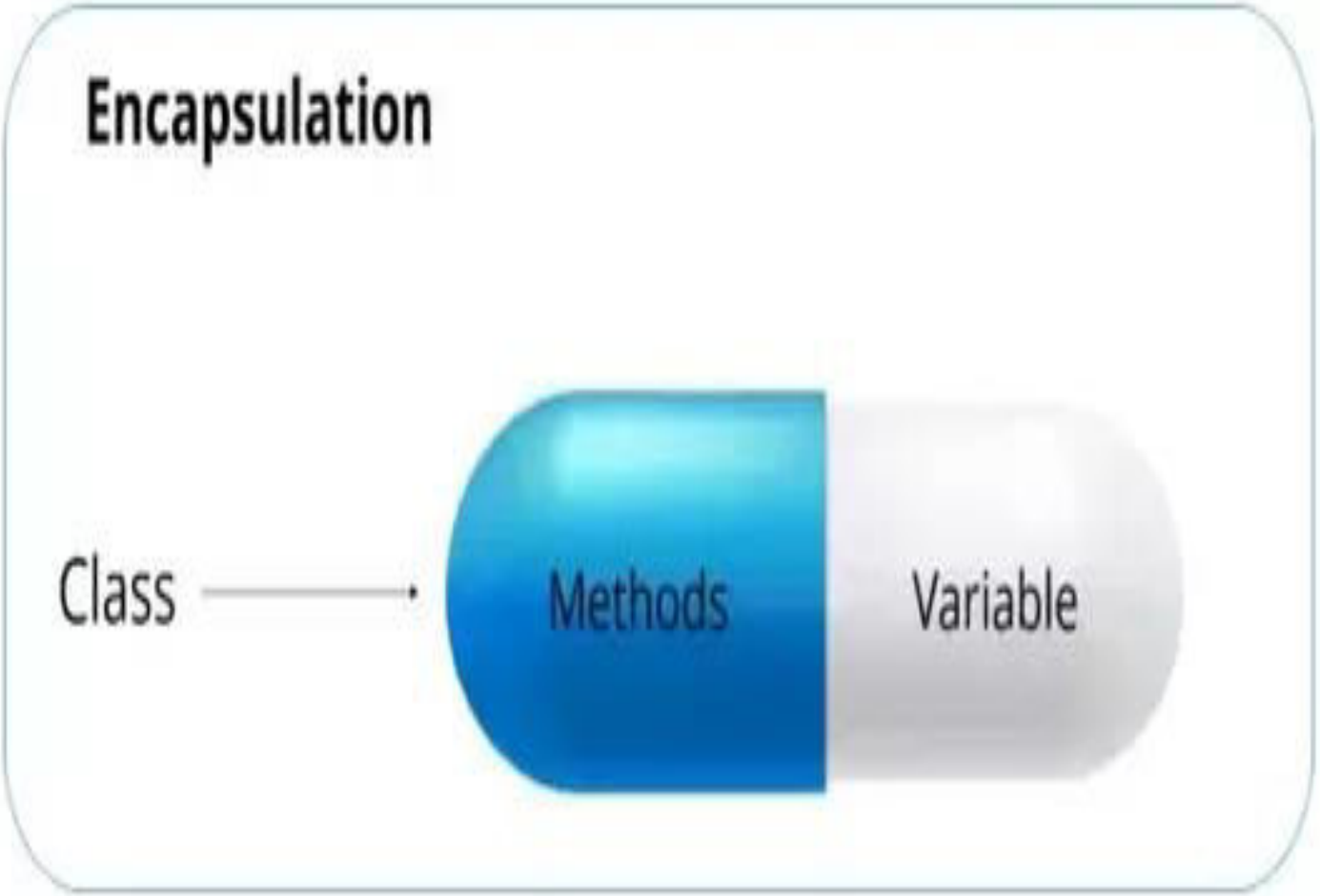

Object
Bicycle
Object's Properties
Gear
Pedal
Speed
Applying Breaks, Changing gears
Object's Methods

DATA ABSTRACTION AND ENCAPSULATION

- Wrapping up of data and function in to a single unit (called class) is known as encapsulation
- The data is not accessible to the outside world
- Only the function can access it
- The prevention of the access of data by the outside world is called data hiding or information hiding.
- Abstraction is the act of representing essential features without including the background details

DATA ABSTRACTION

- Data abstraction is a Simplified View of an object that includes only features one is interested in while hides away the unnecessary details.


Encapsulation
Class
Methods
Variable

Abstraction	Encapsulation
Abstraction is a general concept formed by extracting common features from specific examples or The act of withdrawing or removing something **unnecessary**.	Encapsulation is the mechanism that binds together code and the data it manipulates, and keeps both **safe from outside interference** and **misuse**.
You can use abstraction using **Interface** and **Abstract** Class	You can implement encapsulation using **Access Modifiers** (Public, Protected & Private)
Abstraction solves the problem in **Design** Level	Encapsulation solves the problem in **Implementation** Level
For simplicity, abstraction means hiding implementation using Abstract class and Interface	For simplicity, encapsulation means hiding data using getters and setters

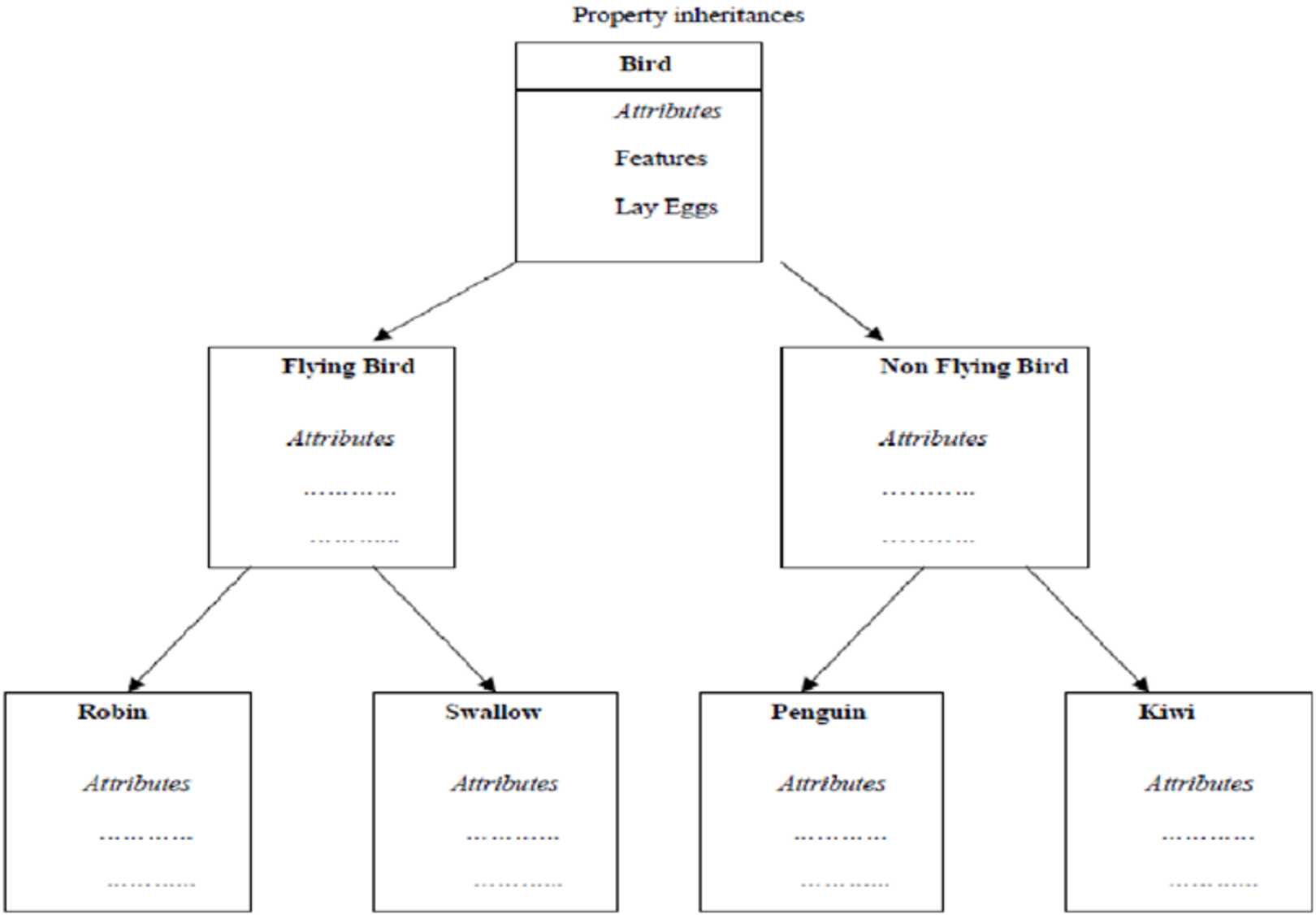
INHERITANCE

- Objects of one class acquire the properties of another classes
- It supports the concept of hierarchical classification
- Inheritance provides the idea of reusability
  - ie We can add additional features to an existing class without modifying it
  - By deriving a new class from the existing one

INHERITANCE

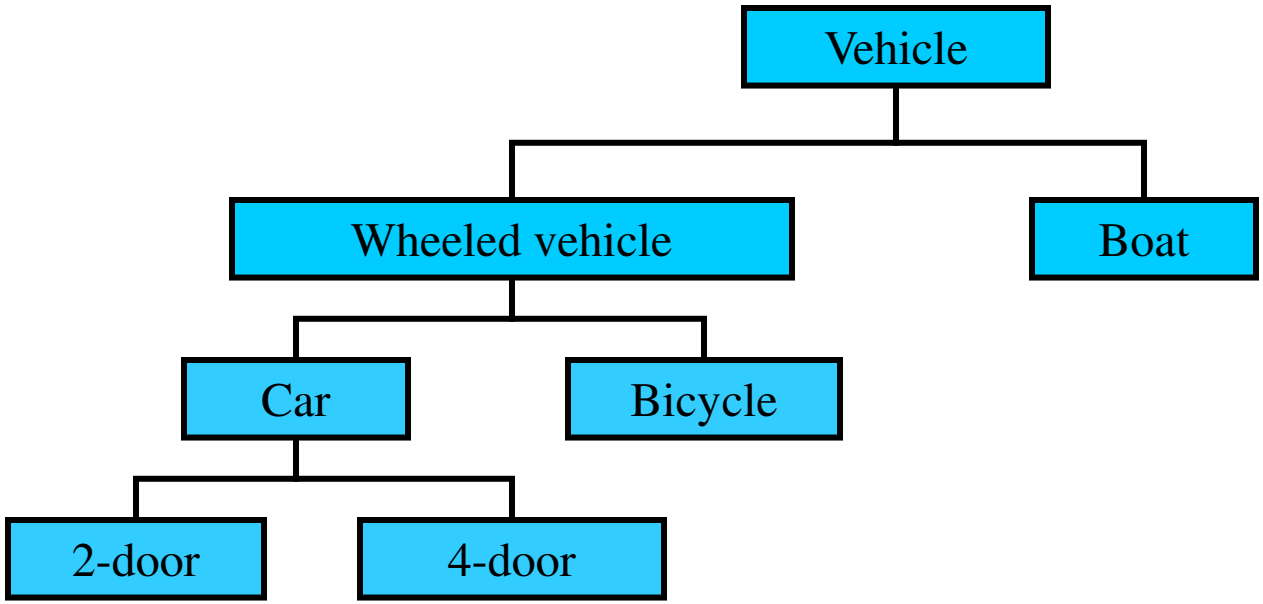
- The language mechanism by which one class acquires the properties (data and operations /methods) of another class
- **Base Class (or superclass):** the class being inherited from
- **Derived Class (or subclass):** the class that inherits

INHERITANCE



ARRANGE CONCEPTS INTO AN INHERITANCE HIERARCHY

- Concepts at higher levels are more general
- Concepts at lower levels are more specific (inherit properties of concepts at higher levels)



ADVANTAGES OF INHERITANCE

- When a class inherits from another class, there are **three** benefits:
- (1) You can *reuse* the methods and data of the existing class

  (2) You can *extend* the existing class by adding new data and new methods

  (3) You can *modify* the existing class by overloading its methods with your own implementations

INHERITANCE AND ACCESSIBILITY

- A class inherits the ***behavior*** of another class and enhances it in some way
- Inheritance ***does not*** mean inheriting access to another class' private members

RULES FOR BUILDING A CLASS HIERARCHY

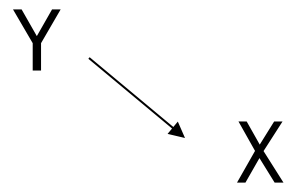
- Derived classes are **special cases** of base classes
- A derived class **can also serve** as a base class for new classes.
- There is no limit on the **depth of inheritance** allowed in C++ (as far as it is within the limits of your compiler)
- It is possible for a class to be a base class for **more than one** derived class

PROTECTED CLASS MEMBERS

- Derived classes **cannot** access the private data of the base class
- Declaring methods and data of the base class as ***protected*** (instead of private) allows derived classes to access them
- Objects outside the class, however, cannot access them (same as private)

PROTECTED AND PRIVATE INHERITANCE

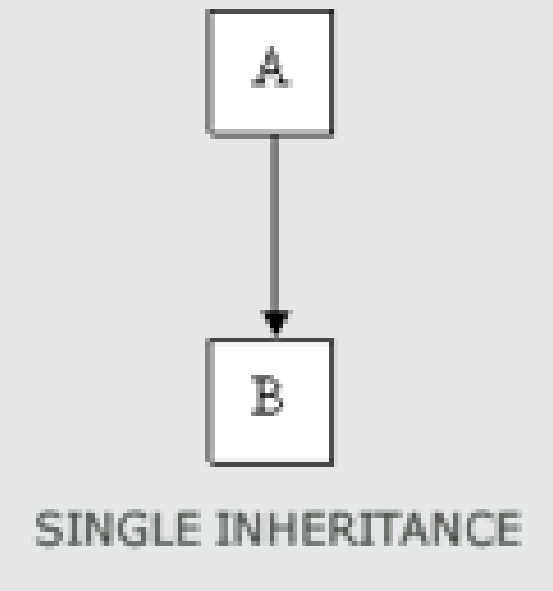
```
class X : protected Y {
   ...
};
```



- With ***protected inheritance***, public and protected members of Y become protected in X (i.e., classes derived from X inherit the public members of Y as protected)
- With ***private inheritance***, public and protected members of Y become private in X (i.e., classes derived from X inherit the public members of Y as private)
- Default inheritance: ***private***

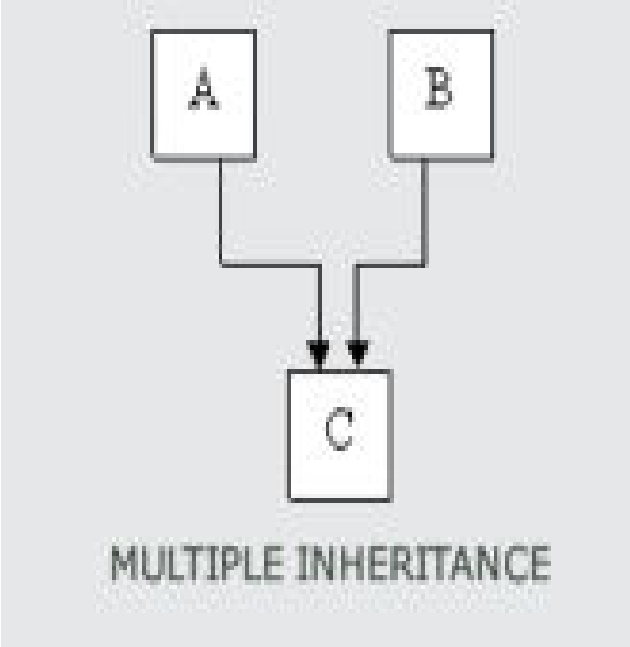
FORMS OF INHERITANCE

- **Single Inheritance:** It is the inheritance hierarchy wherein one derived class inherits from one base class.



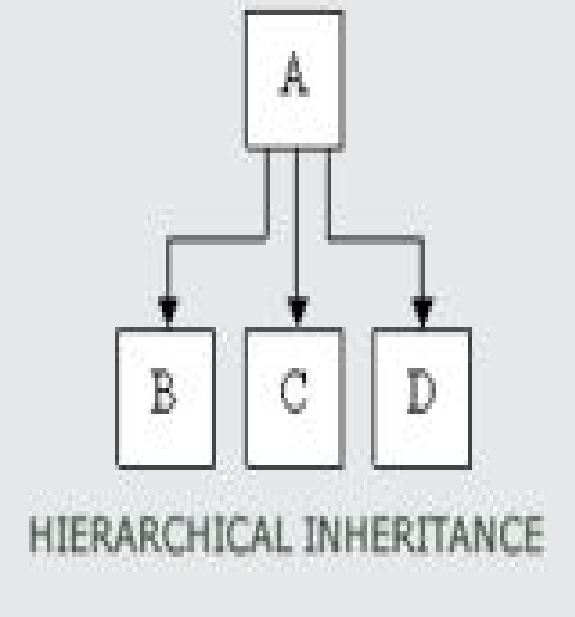
FORMS OF INHERITANCE

- **Multiple Inheritance:** It is the inheritance hierarchy wherein one derived class inherits from multiple base class(es)



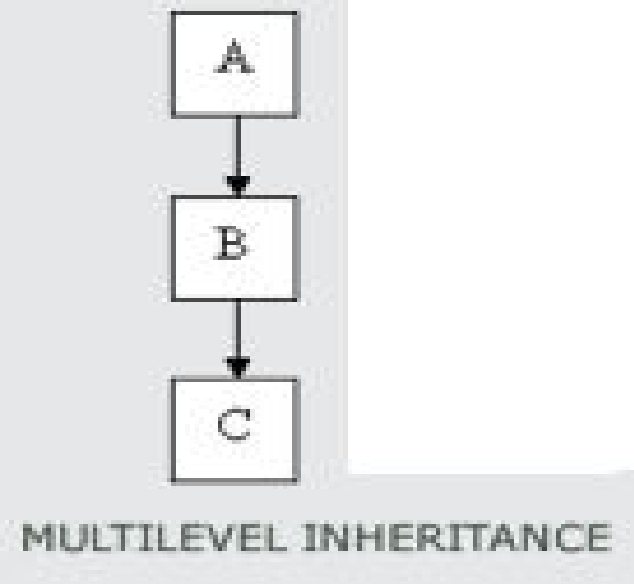
FORMS OF INHERITANCE

- **Hierarchical Inheritance:** It is the inheritance hierarchy wherein multiple subclasses inherit from one base class.



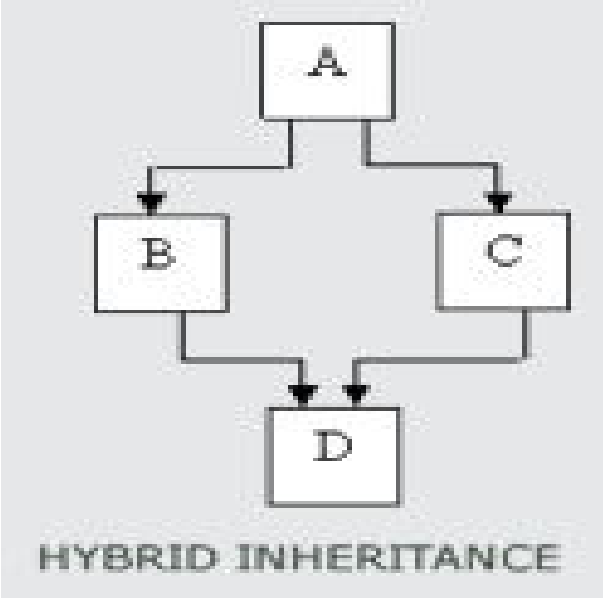
FORMS OF INHERITANCE

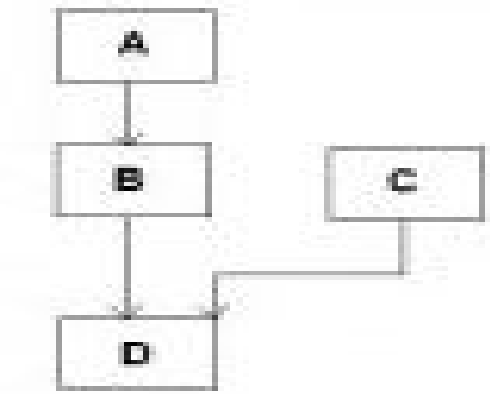
- **Multilevel Inheritance:** It is the inheritance hierarchy wherein subclass acts as a base class for other classes.



FORMS OF INHERITANCE

- **Hybrid Inheritance:** The inheritance hierarchy that reflects any legal combination of other four types of inheritance.





MEMBER ACCESS SPECIFIER

Member Access Specifier	How Members of the Base Class Appear in the Derived Class
Private	Private members of the base class are inaccessible to the derived class.
	Protected members of the base class become private members of the derived class.
	Public members of the base class become private members of the derived class.
Protected	Private members of the base class are inaccessible to the derived class.
	Protected members of the base class become protected members of the derived class.
	Public members of the base class become protected members of the derived class.
Public	Private members of the base class are inaccessible to the derived class.
	Protected members of the base class become protected members of the derived class.
	Public members of the base class become public members of the derived class.

A derived class inherits every member of a base class except constructor and destructor

SUMMARIZATION OF ACCESS TYPE

Access	Public	Protected	Private
Same Class	Yes	Yes	Yes
Derived Classes	Yes	Yes	No
Outside Classes	Yes	No	No

TYPE OF INHERITANCE

- Public Inheritance
- Protected Inheritance
- Private Inheritance

PUBLIC INHERITANCE

- Public members of base class will be public members in derived class
- Protected members of the base class become protected members of the derived class
- Private members are never accessible directly from a derived class but can be accessed through calls to the public and protected members

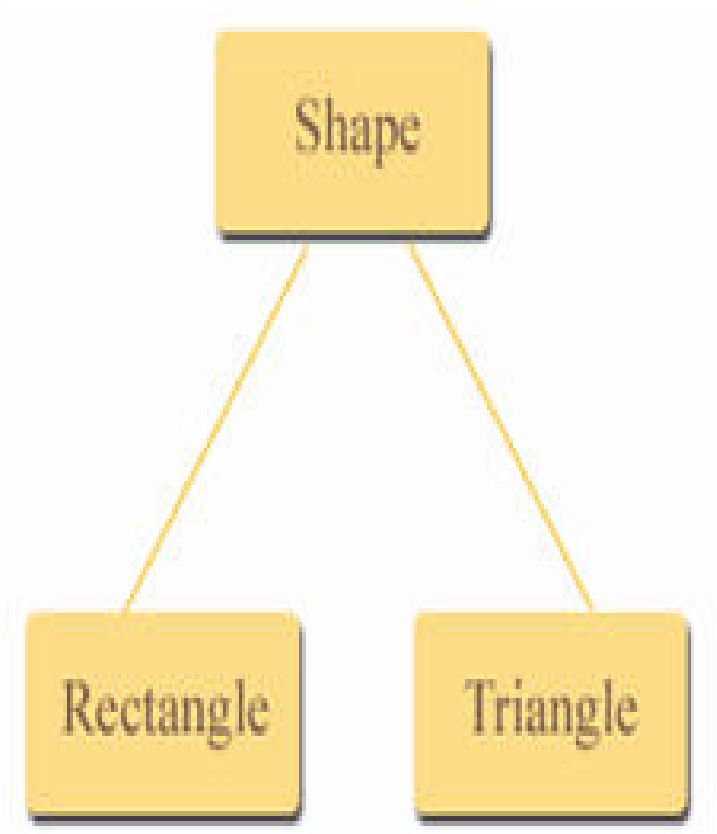
PROTECTED INHERITANCE

- When deriving from the base class public and protected members of the base class become protected members of the derived class

PRIVATE INHERITANCE

- When deriving from a private base class, public and protected members of the base class become private members of the derived class

EXAMPLE: INHERITANCE



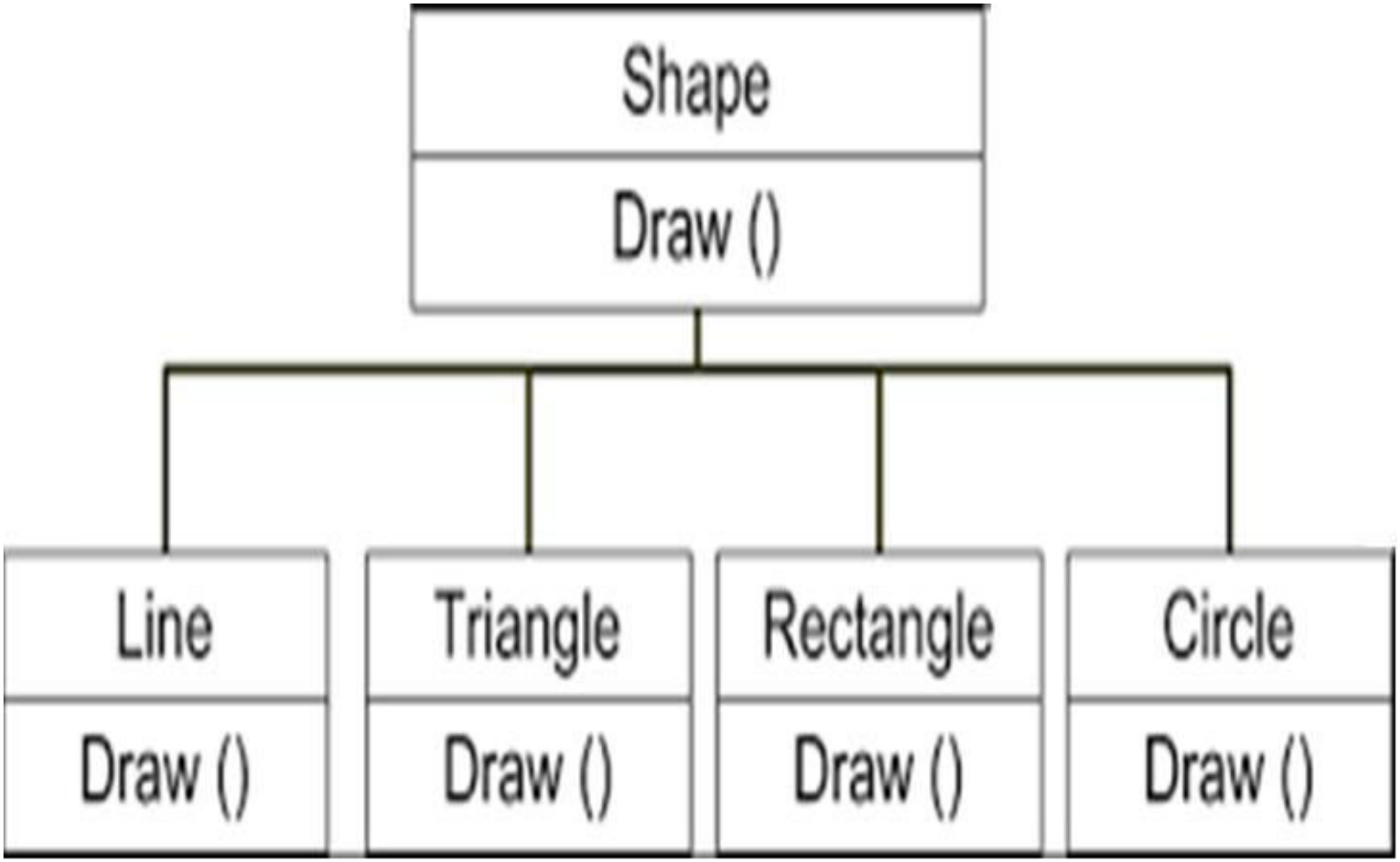
POLYMORPHISM

- It s a GREEK word, means the ability to take more than one form
- “Poly mean Multiple” + “Morph means Forms”
- An operation may exhibit different behavior in different instances
- The behavior depends on the types of data used in the operation.
- The process of making an operator to exhibit different behaviors in different type of task is known as *operator overloading*

POLYMORPHISM

- Single function name to perform different type of task is known as function overloading
- Polymorphism is extensively used in implementing inheritance.

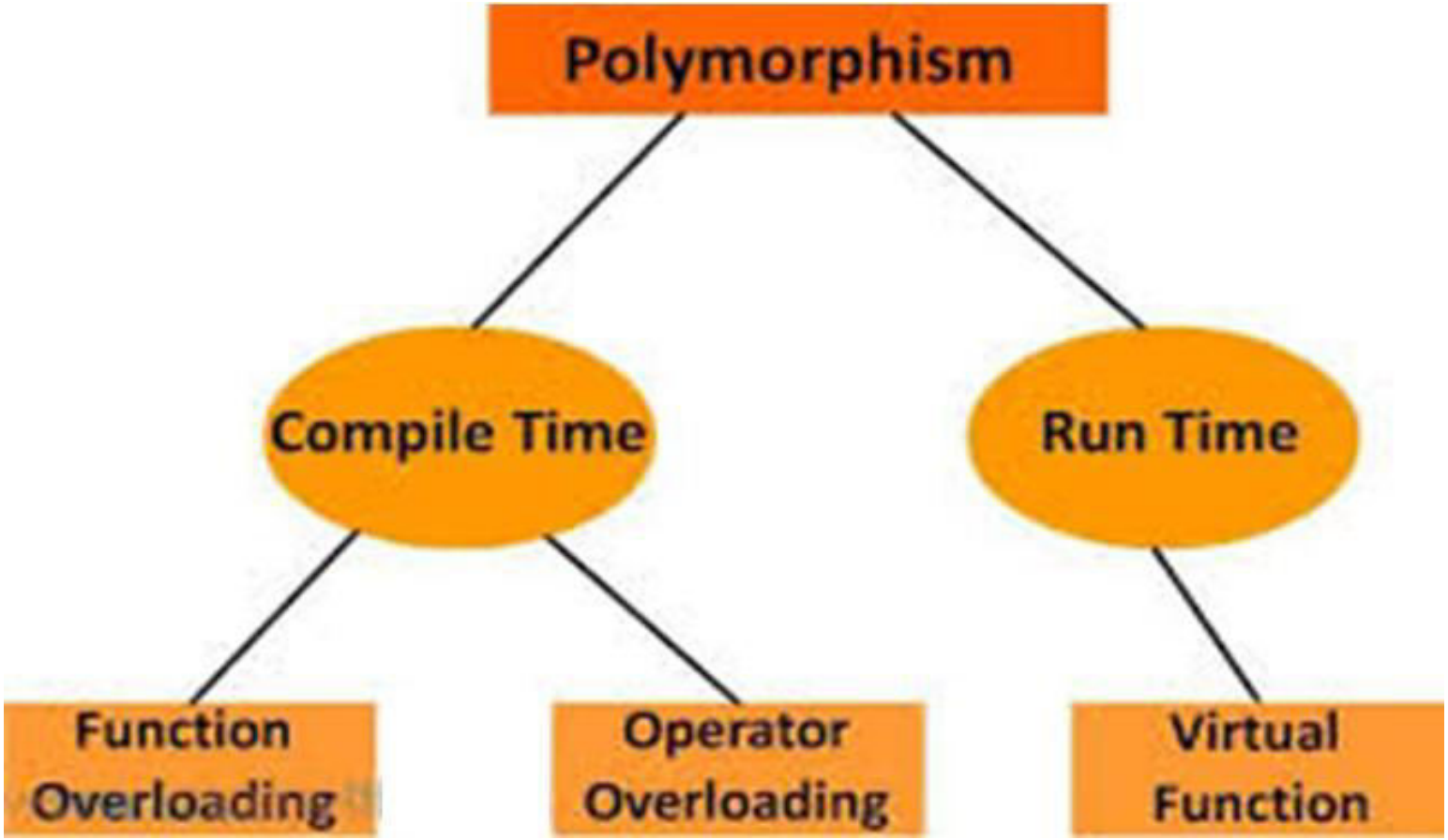
POLYMORPHISM



OVERLOADING VS. OVERRIDING

- Don't confuse the concepts of overloading and overriding
- Overloading deals with multiple methods in the same class with the same name but different signatures
- Overriding deals with two methods, one in a parent class and one in a child class, that have the same signature
- Overloading allows you define a similar operation in different ways for different data
- Overriding allows you define a similar operation in different ways for different object types

POLYMORPHISM



Compile time Polymorphism	Run time Polymorphism
In Compile time Polymorphism, call is resolved by the **compiler**.	In Run time Polymorphism, call is **not** resolved by the compiler.
It is also known as **Static binding, Early binding** and **overloading** as well.	It is also known as **Dynamic binding, Late binding** and **overriding** as well.
Overloading is compile time polymorphism where more than one methods share the same name with different parameters or signature and different return type.	**Overriding** is run time polymorphism having same method with same parameters or signature, but associated in a class & its subclass.
It is achieved by **function** overloading and **operator** overloading.	It is achieved by **virtual functions** and **pointers**.
It provides **fast execution** because known early at compile time.	It provides **slow execution** as compare to early binding because it is known at runtime.
Compile time polymorphism is **less flexible** as all things execute at compile time.	Run time polymorphism is **more flexible** as all things execute at run time.

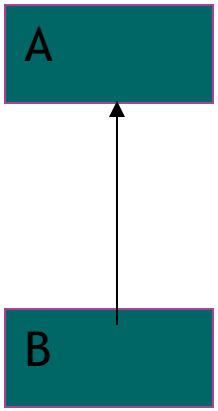
OBJECT-ORIENT DESIGN (OOD)

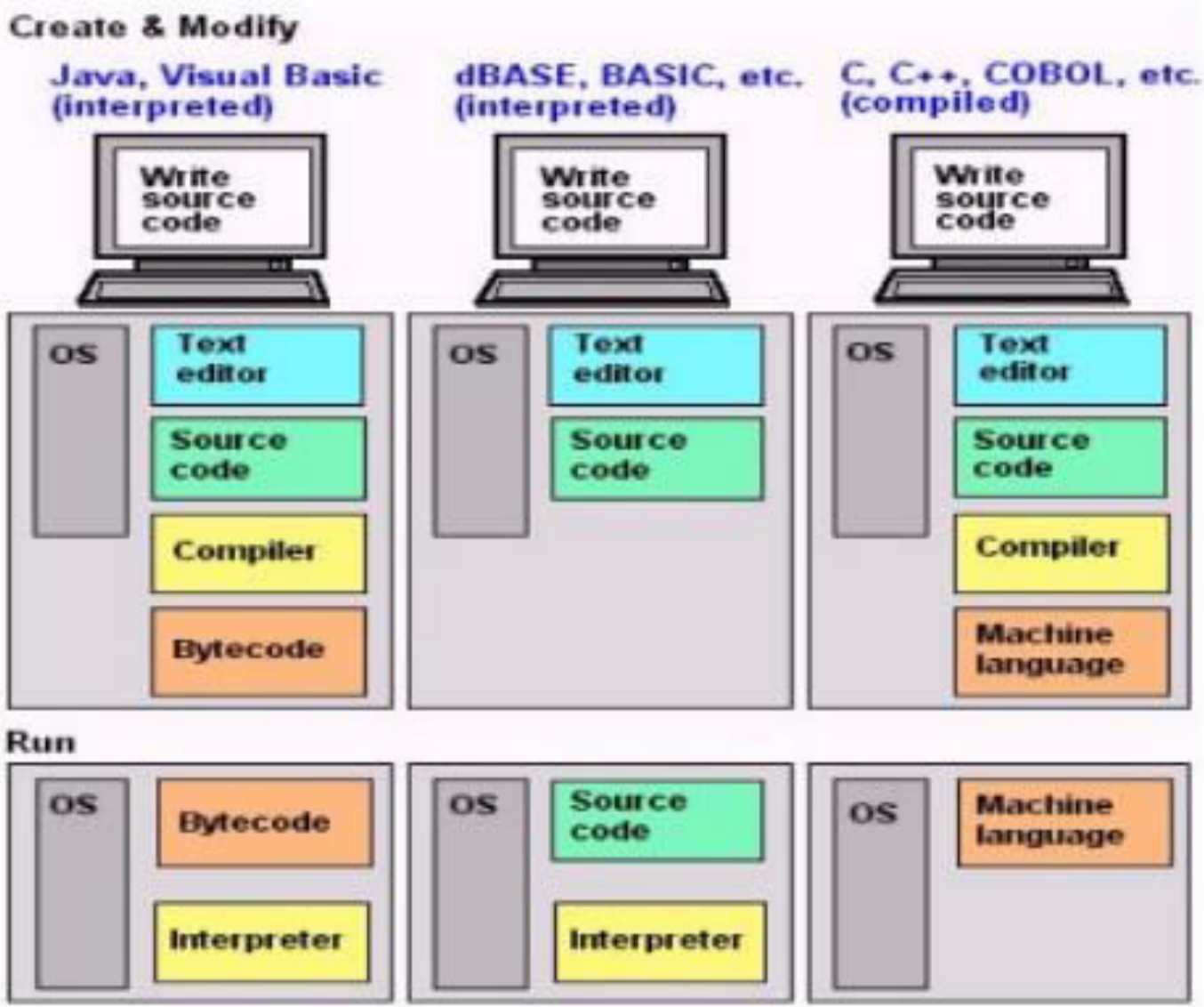
- Identify the problem's objects
  - if an object cannot be represented by an existing type, design a class to do so
  - if two or more classes share common attributes, design a hierarchy
- Identify the operations, If an operation cannot be performed with an existing operator or method
  - define a method to do so
  - store the method within a class hierarchy to enable inheritance
- Organize the objects and operations into an algorithm

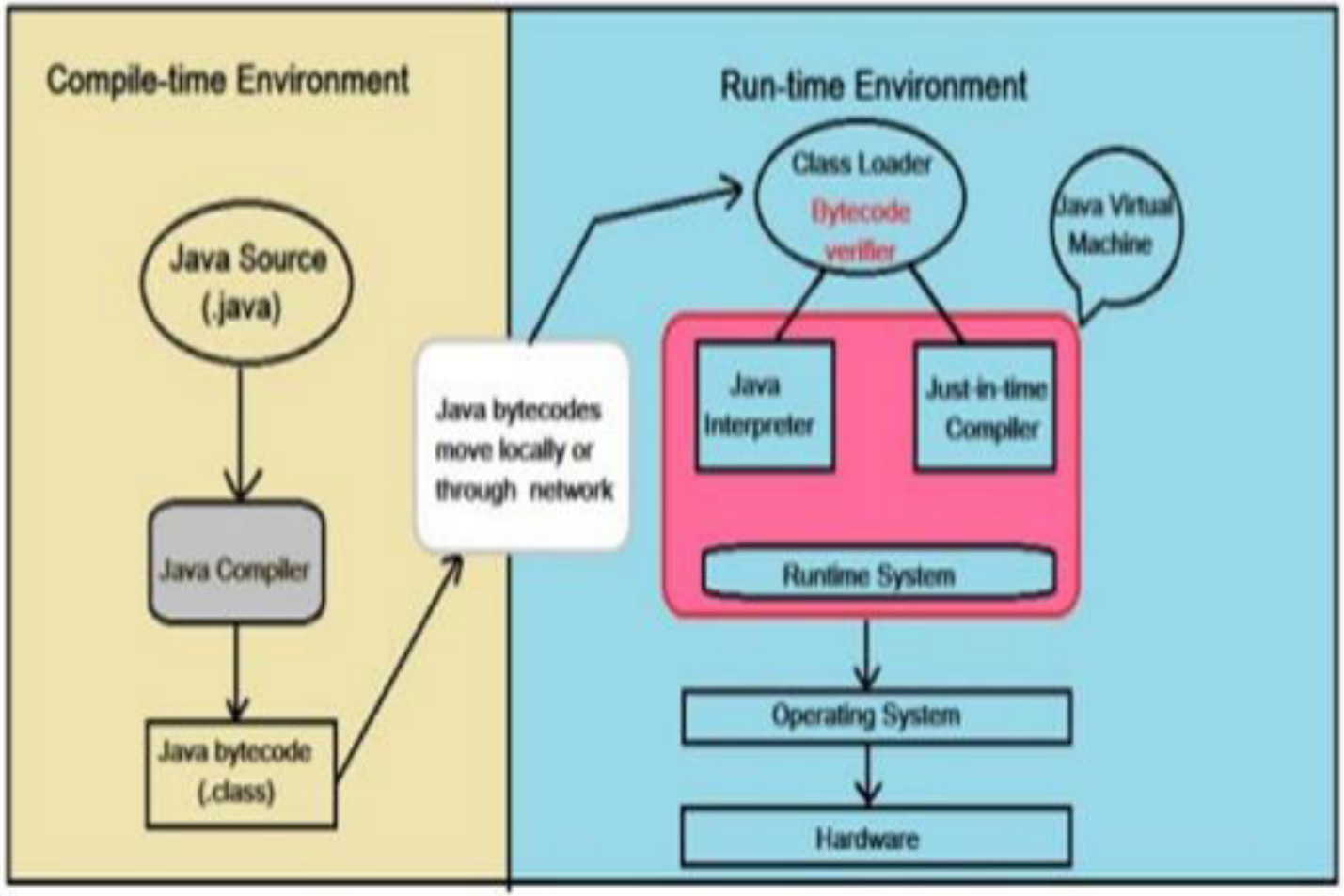
O-O DESIGN ISSUES

Using the extends relationship: A class **B** should extend another class **A** if and only if

- **B** "is a" specialized version of **A** and …
- All messages that can be sent to **A** can be appropriately sent to **B**




Create & Modify
Java, Visual Basic (interpreted)
dBASE, BASIC, etc. (interpreted)
C, C++, COBOL, etc. (compiled)
Write source code
Write source code
Write source code
OS
Text editor
Source code
Compiler
Bytecode
OS
Text editor
Source code
OS
Text editor
Source code
Compiler
Machine language
Run
OS
Bytecode
Interpreter
OS
Source code
Interpreter
OS
Machine language


Compile-time Environment
Java Source
(.java)
Java Compiler
Java bytecode
(.class)
Java bytecodes
move locally or
through network
Run-time Environment
Class Loader
Bytecode
verifier
Java Virtual
Machine
Java
Interpreter
Just-in-time
Compiler
Runtime System
Operating System
Hardware