CSE 223: Programming -2
01-Introduction

Prof. Dr. Khaled Nagi

Department of Computer and Systems Engineering,

Faculty of Engineering, Alexandria University, Egypt.

Agenda

- Welcome
- Course Information
- Brief Introduction to Object Oriented Paradigm
- Basic differences between C++ and JAVA

Organization

- Instructor: Prof. Dr. Khaled Nagi
  - Email: khaled.nagi@alexu.edu.eg
- eLearning Platform: MS-Teams
  - https://teams.microsoft.com/l/team/19%3aOkhV7M-KesUZNIJnXtOm93uyb6z0xFT8UUdRaV6TrGY1%40thread.tacv2/conversations?groupId=d513da0b-dc69-411a-9616-45e20563a7da&tenantId=eaf624c8-a0c4-4195-87d2-443e5d7516cd
  - Joining code: **gk0cl0w**
- Lecture time: Sunday: 1st period
- Practical
  - TBD
- Grading Scheme
  - Mid-Term: 10%: MCQ
  - Class work: 30%: sheets, Programming assignments, group projects
  - End of Term: 60% MCQ
    - Might change depending on CoViD-19 situation

Course Information

- Course Breakdown
  - Object-oriented design
  - Encapsulation and information hiding
  - Separation of behavior and implementation:
    - Classes, Subclasses and inheritance, Polymorphism
  - UML and Requirement analysis
  - Design patterns

Course Materials

- Many Textbooks!
  - Objects, Abstraction, Data Structures and Design Using Java Version 5.0 by Elliot B. Koffman and Paul A. T. Wolfgang
  - The Essence of Object-Oriented Programming with Java and UML by Bruce E. Wampler
  - Object Oriented Programming with Java: Essentials and Applications by Rajkumar Buyya, Thamarai Selvi Somasundaram, and Xingchen Chu
  - Design Patterns in Java by Steven John Mestker and William C. Wake
- Further readings
  - Clean Code
  - Clean Coder
- And More Online Materials
  - You can google any design pattern you want!

This is **NOT** …

- … a Java Course
  - Java Vs. Object Oriented Programming
  - Thinking Objects
  - Object Oriented Programming Principles
- … Just About Programming
  - Software Development is much more than just programming.
  - Object Oriented Analysis and Design Principles
  - Advanced Topics: Design Patterns and Refactoring

Course Contents

- 01 – Introduction
- 02 – UML (I)
- 03 – UML (II)
- 04 – Design Styles and - Basic Design Patterns
- 05 – Creational Design Patterns (I)
- 06 – Creational Design Patterns (II)
- 07 – Partitioning patterns
- 08 – Structural patterns (I)
- 09 – Structural patterns (II)
- 10 – Behavioral patterns (I)
- 11 – Behavioral patterns (II)
- 12 – Concurrency patterns

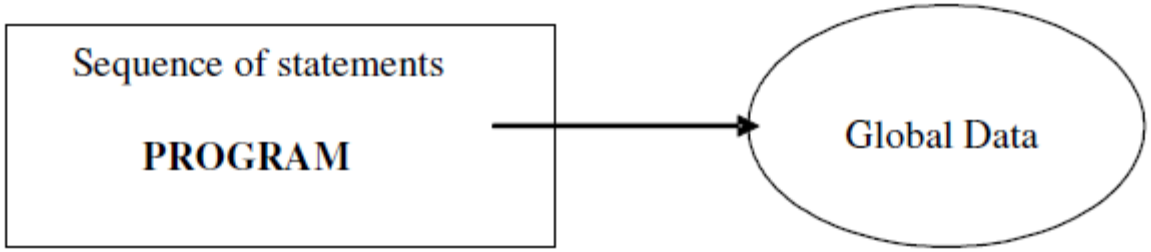
Historical Review

Overview of Programming Paradigms

- Non-Procedural
- Procedural
- Structured
- Object Oriented
- Aspect Oriented

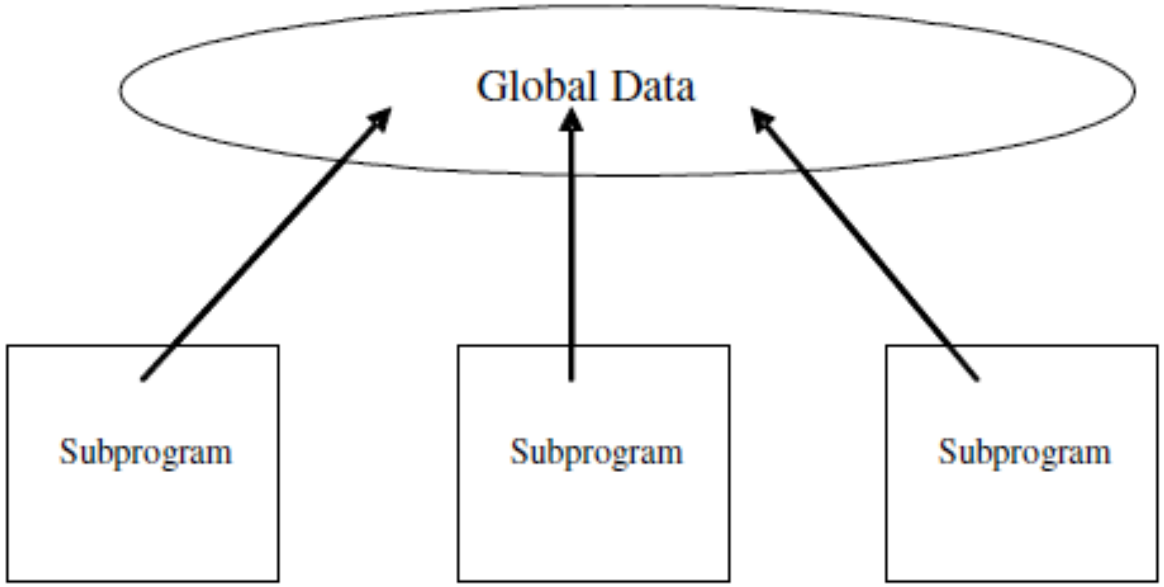
Non-Procedural

- BASIC



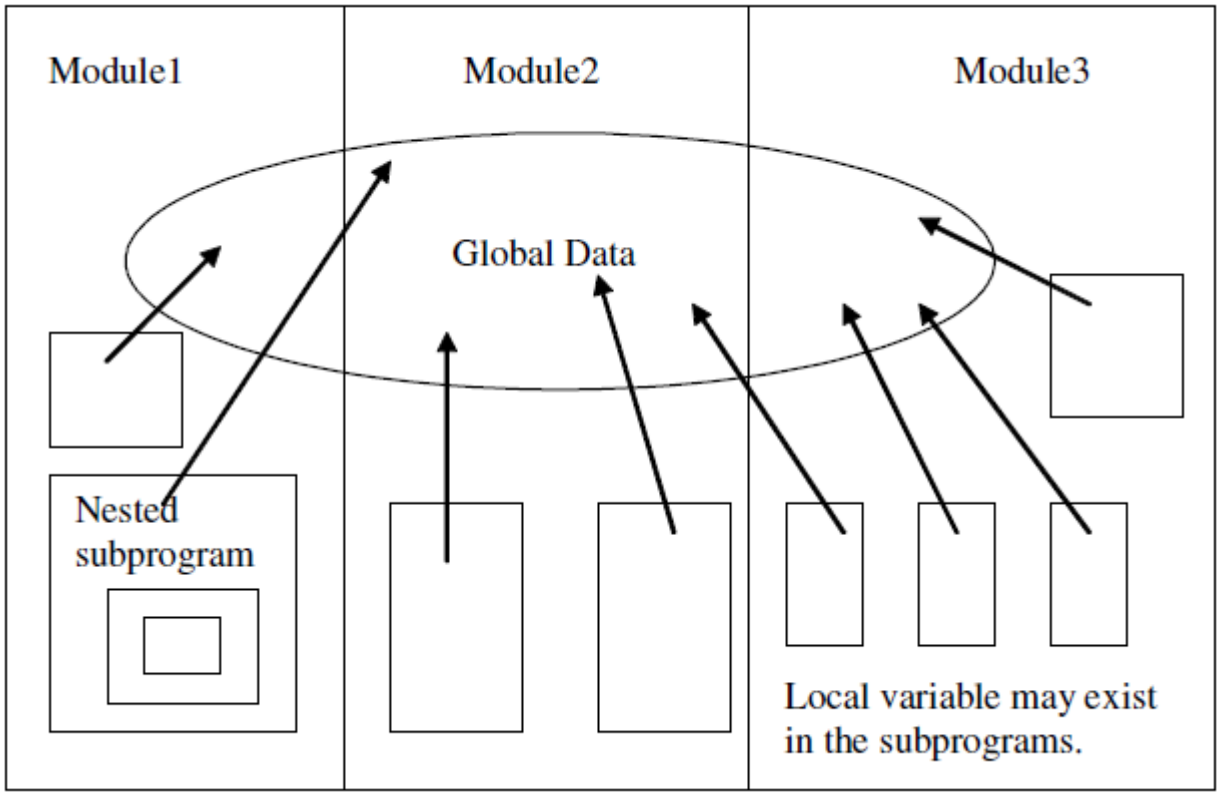
Procedural Programming

- E.g., C, FORTRAN



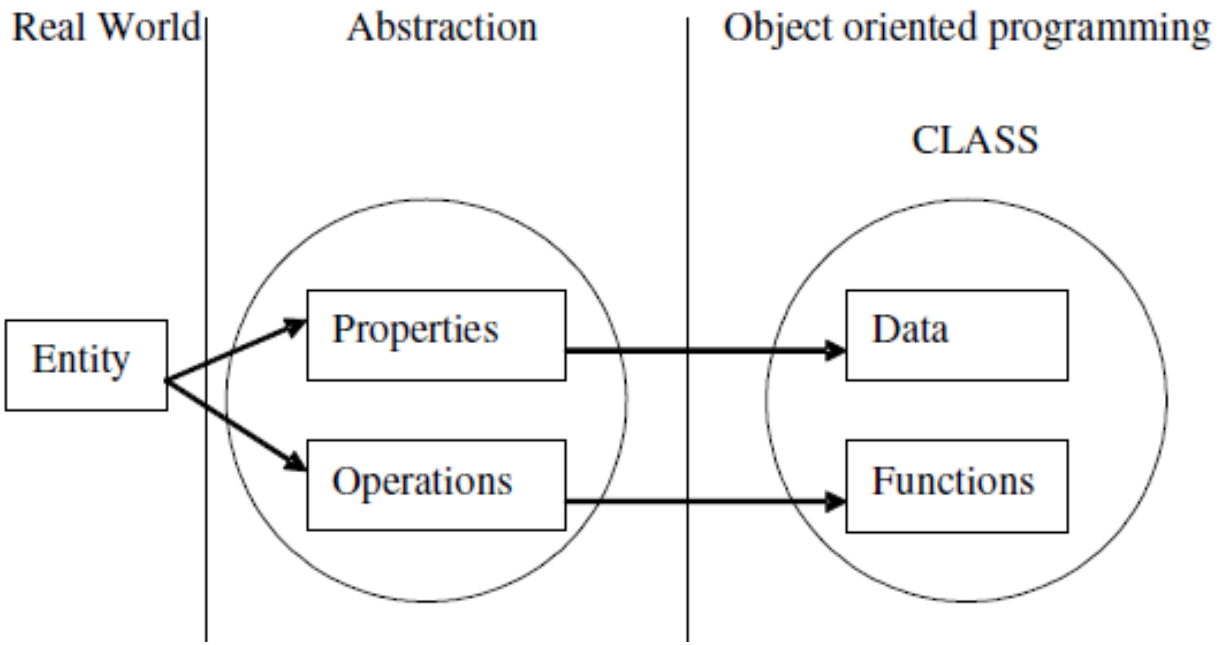
Structured Programming

- E.g., PASCAL



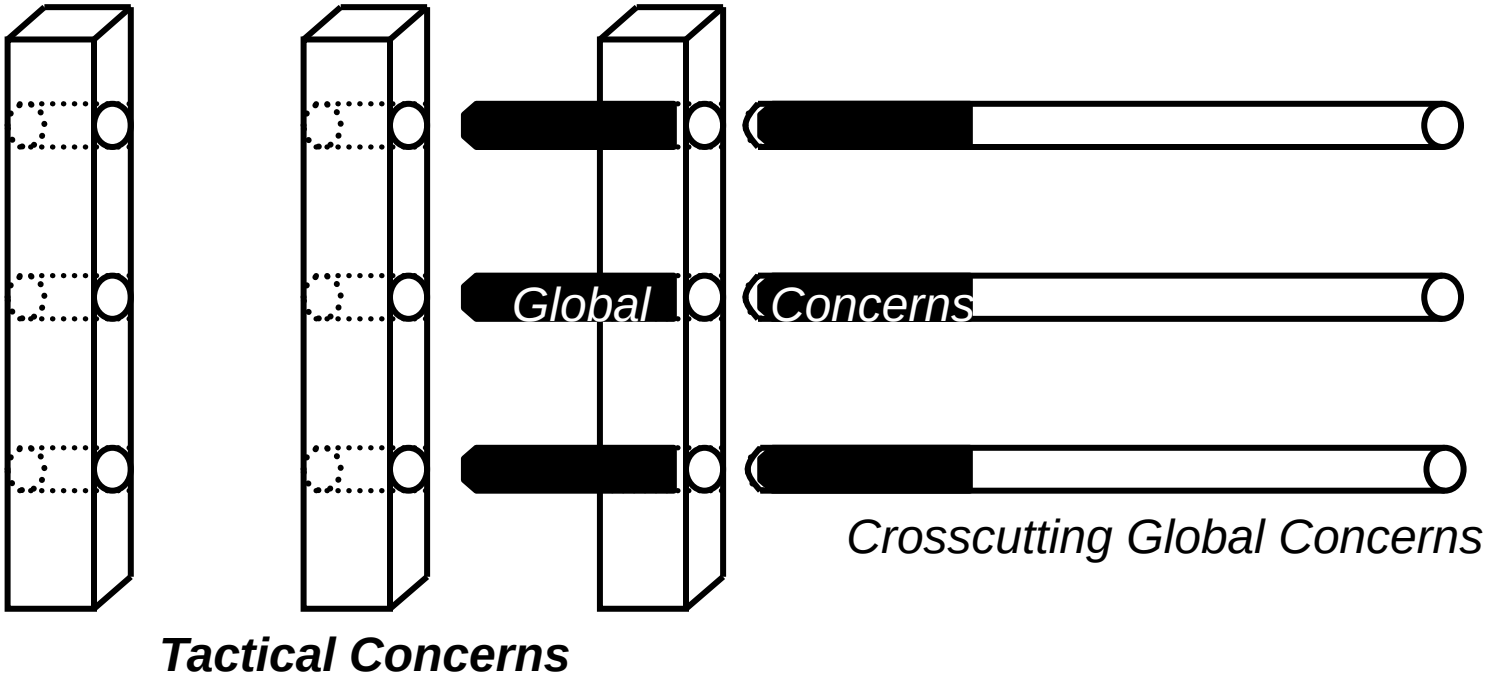
Object Oriented Programming

- Java, c++, c#



Aspect Oriented Programming

- E.g., AspectJ
- For global concerns such as logging, transaction management,



Introduction to Object Oriented Paradigm

Advantages of OOP

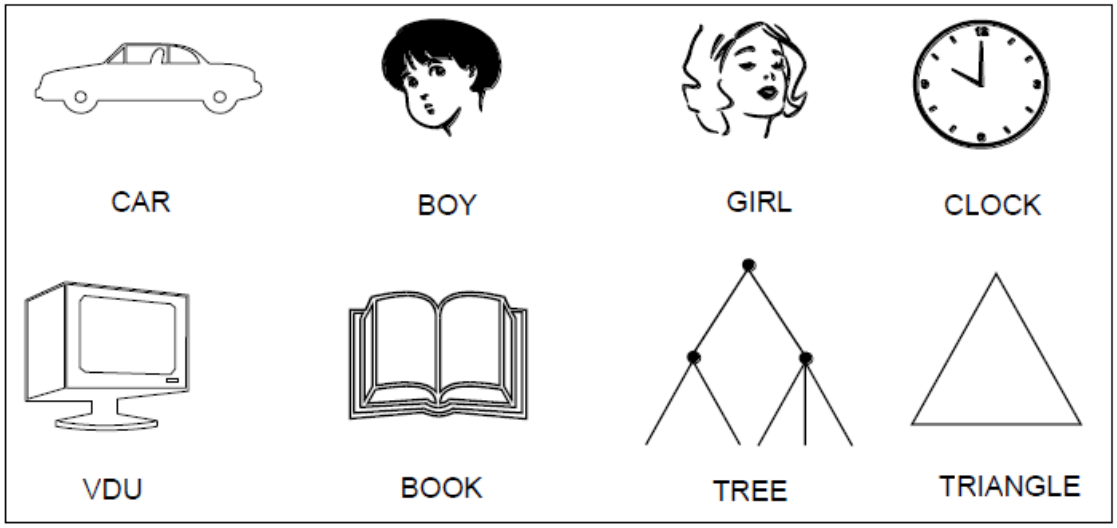
- **Simplicity**: software objects model real world objects, so the complexity is reduced, and the program structure is very clear;
- **Modularity**: each object forms a separate entity whose internal workings are decoupled from other parts of the system;
- **Modifiability**: it is easy to make minor changes in the data representation or the procedures in an OO program. Changes inside a class do not affect any other part of a program, since the only public interface that the external world has to a class is through the use of methods;
- **Extensibility**: adding new features or responding to changing operating environments can be solved by introducing a few new objects and modifying some existing ones;
- **Maintainability**: objects can be maintained separately, making locating and fixing problems easier;
- **Re-usability**: objects can be reused in different programs.

Basic Concepts of OOP

- Classes and Objects
- The Concept of Abstraction
- The Concept of Encapsulation
- The Concept of Generalization/Specialization
  - Inheritance
  - Multiple Inheritance
- The Concept of Polymorphism
  - Overriding
  - Overloading

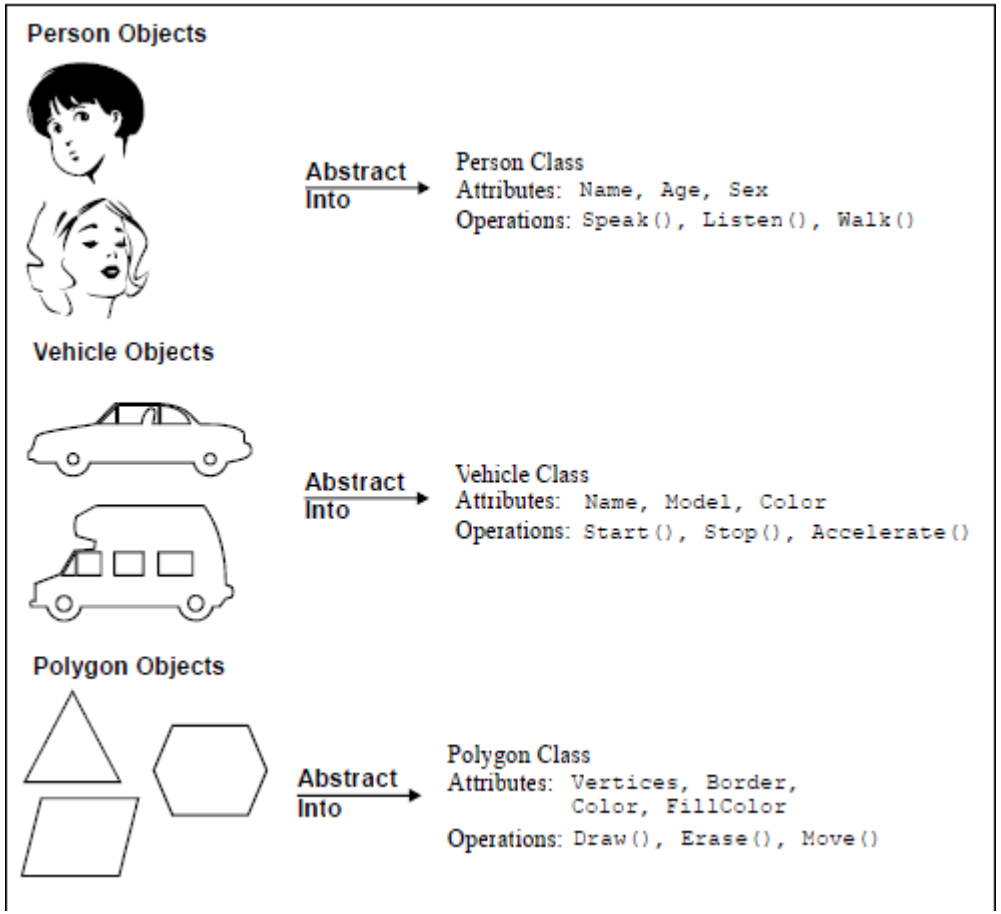
Objects and Classes

- What is an **Object**?
- "Concept, abstraction, or thing with crisp boundary & meaning for a problem"
  - An Object has state and behavior
  - Objects receive stimuli/messages & respond
  - Receiving a stimulus, Object may change state
- Examples of Objects



Objects and Classes

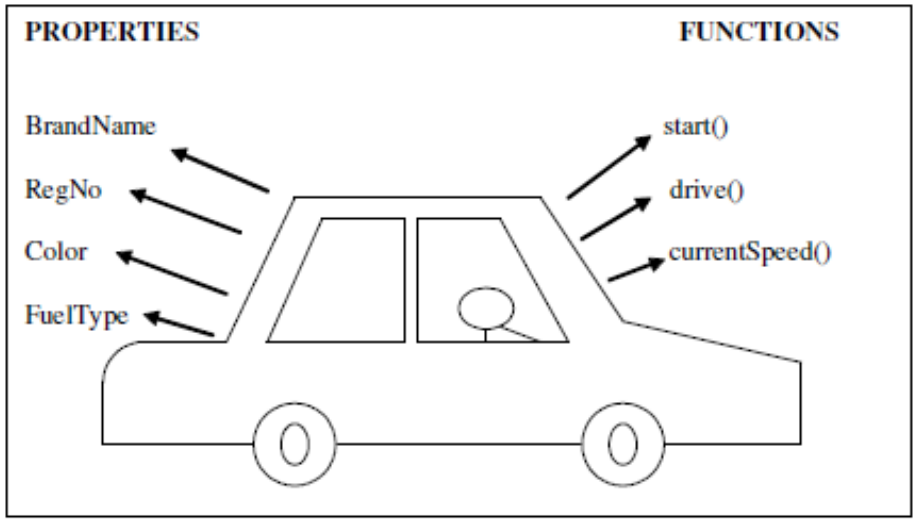
- What is a **Class**?
- Group of Objects with similar
  - properties (attributes)
  - behavior
  - relationships to other objects
  - semantics
- Blueprints of Objects

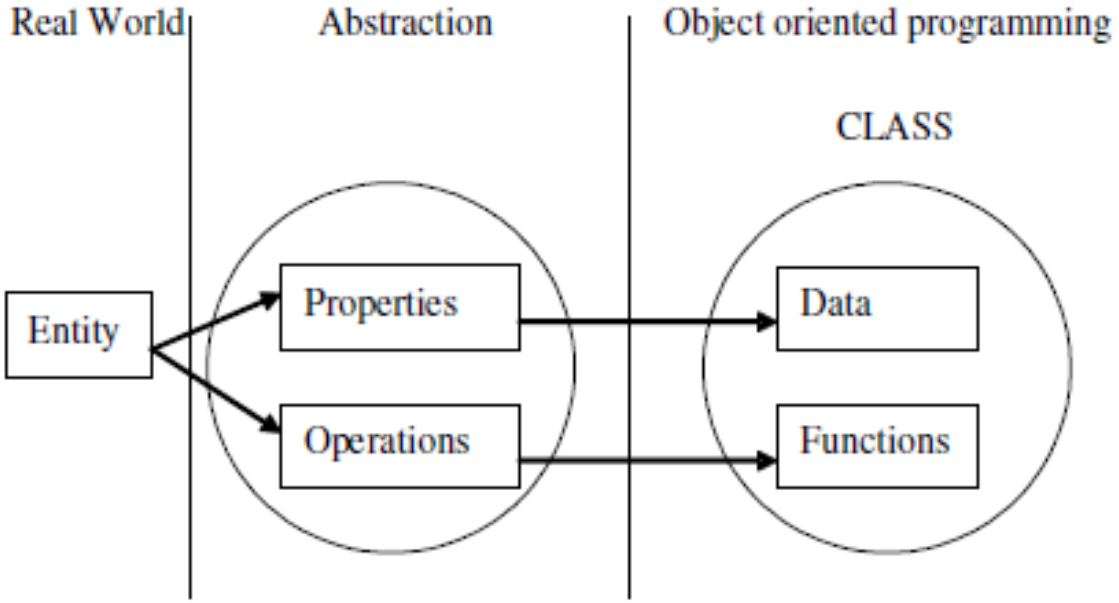


Abstraction

- An abstraction is a view or **representation** of an entity that includes only the most significant attributes.
- *A simplified description of a system that emphasizes some of the system's details ... while suppressing others*
- *The process of reducing an object to its essence so that only the necessary elements are represented. Abstraction defines an object in terms of its **properties** (attributes), **behaviors** (functionality), and **interface** (means of communicating with other objects)*
- Nearly all programming languages since 1980 support data abstraction
- … please refer to our Abstract Data Types (ADTs)

Abstraction Example

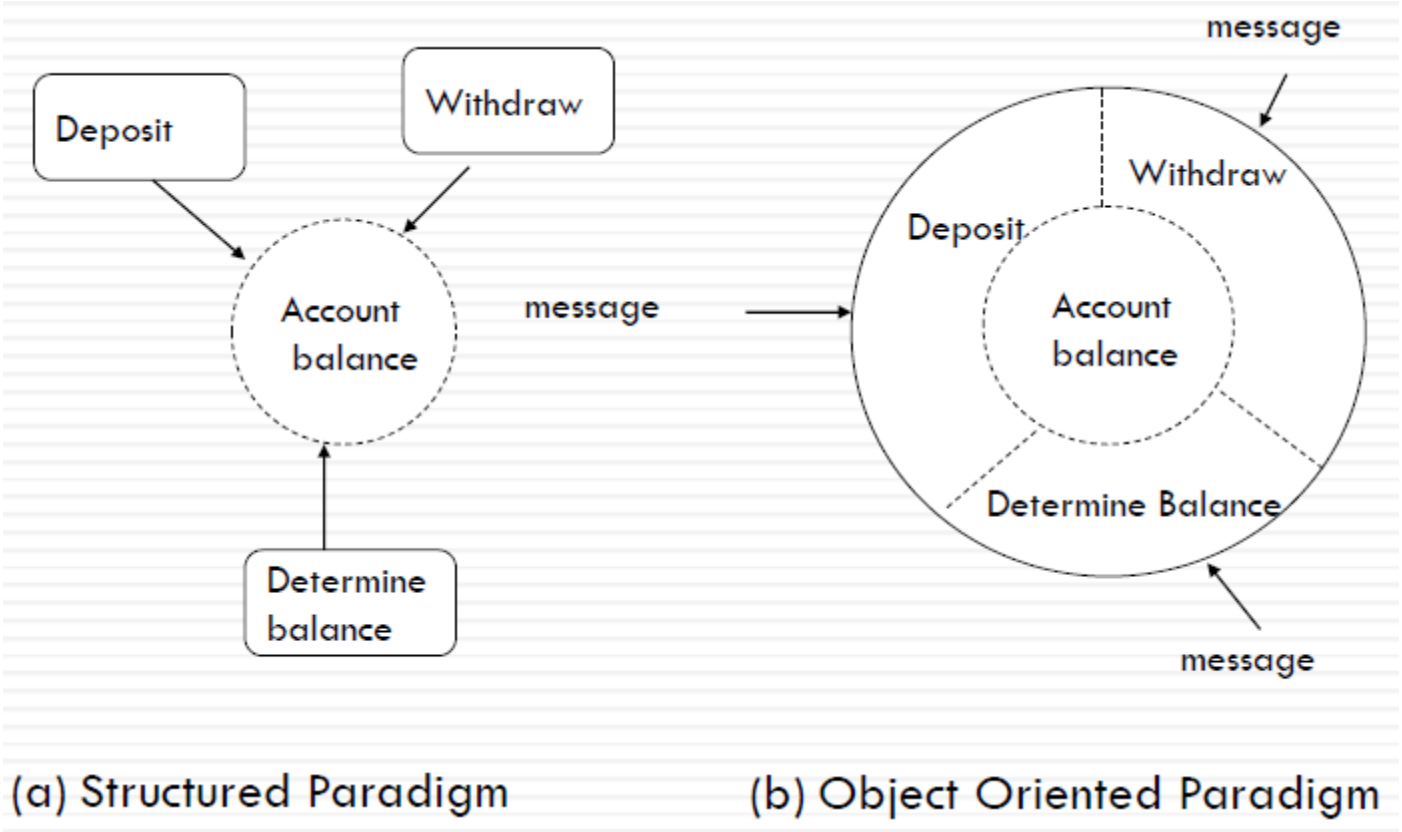




Encapsulation

- *The technique of hiding the internal implementation detail of an object from its external views. Internal structure remains private and services can be accessed by other objects only through messages passed via a clearly defined interface. Encapsulation ensures that the object providing service can prevent other objects from manipulating its data or procedures directly, and it enables the object requesting service to ignore the details of how that service is provided.*
- Information hiding
- Interface Implementation
- Behavior & Data

Example

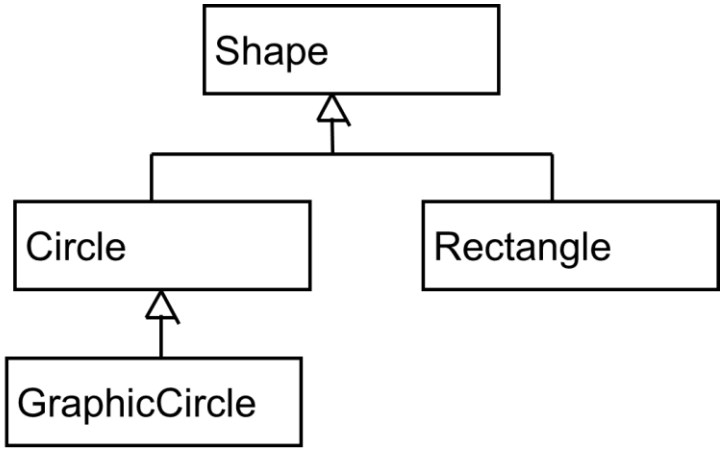

Deposit
Withdraw
Account balance
message
Determine balance
message
Withdraw
Deposit
Account balance
Determine Balance
message
(a) Structured Paradigm
(b) Object Oriented Paradigm

Inheritance

- Object oriented programming allows classes to **inherit** commonly used state and behavior from other classes.
- Software reusability
- Create new class from existing class
  - Absorb existing class's data and behaviors
  - Enhance with new capabilities
- Super-classes and Subclasses
  - Subclass
    - More specialized group of objects
    - Behaviors inherited from superclass
    - Additional behaviors
- Object of one class "**is a**" object of another class
  - Example: Rectangle **is a** quadrilateral.

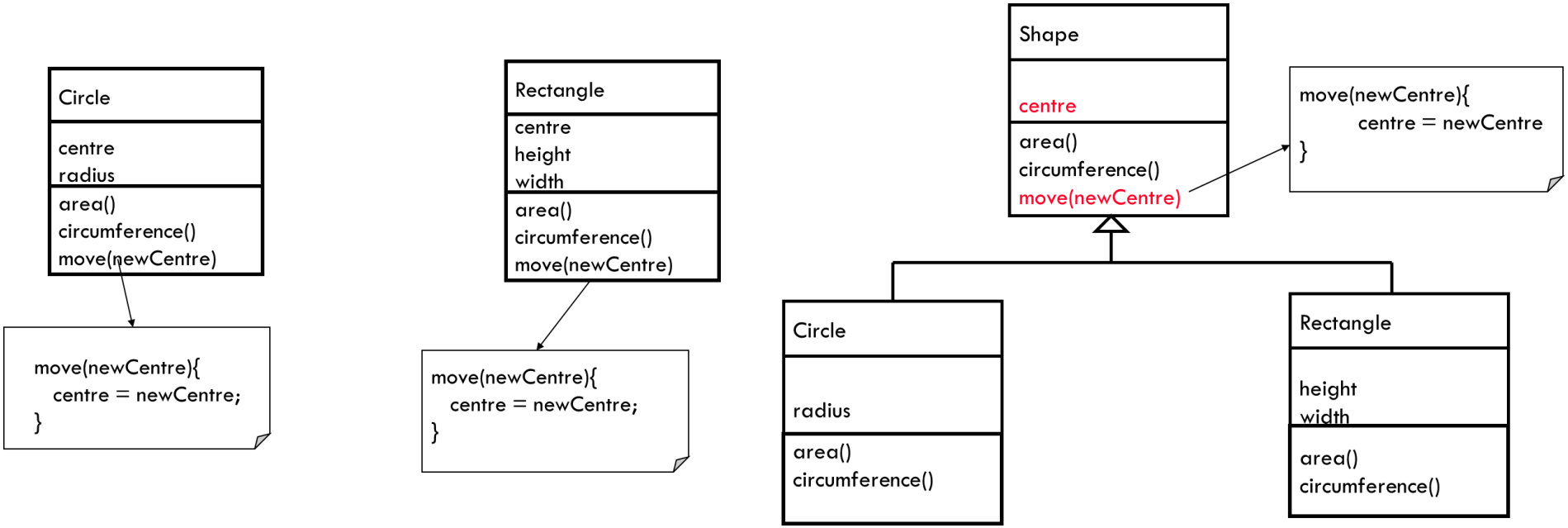
Inheritance Example

- Define **Person** to be a class
  - A Person has **attributes:** age, height, gender
  - Assign values to attributes when describing object
- Define **Student** to be a **subclass** of Person
  - A Student has all attributes of Person, plus attributes of his/her own (student no, course_enrolled)
- A Student **inherits** all attributes of Person
- Define Lecturer to be a subclass of Person
  - A Lecturer has all attributes of Person, plus attributes of his/her own (staff_id, subjectID1, subjectID2)

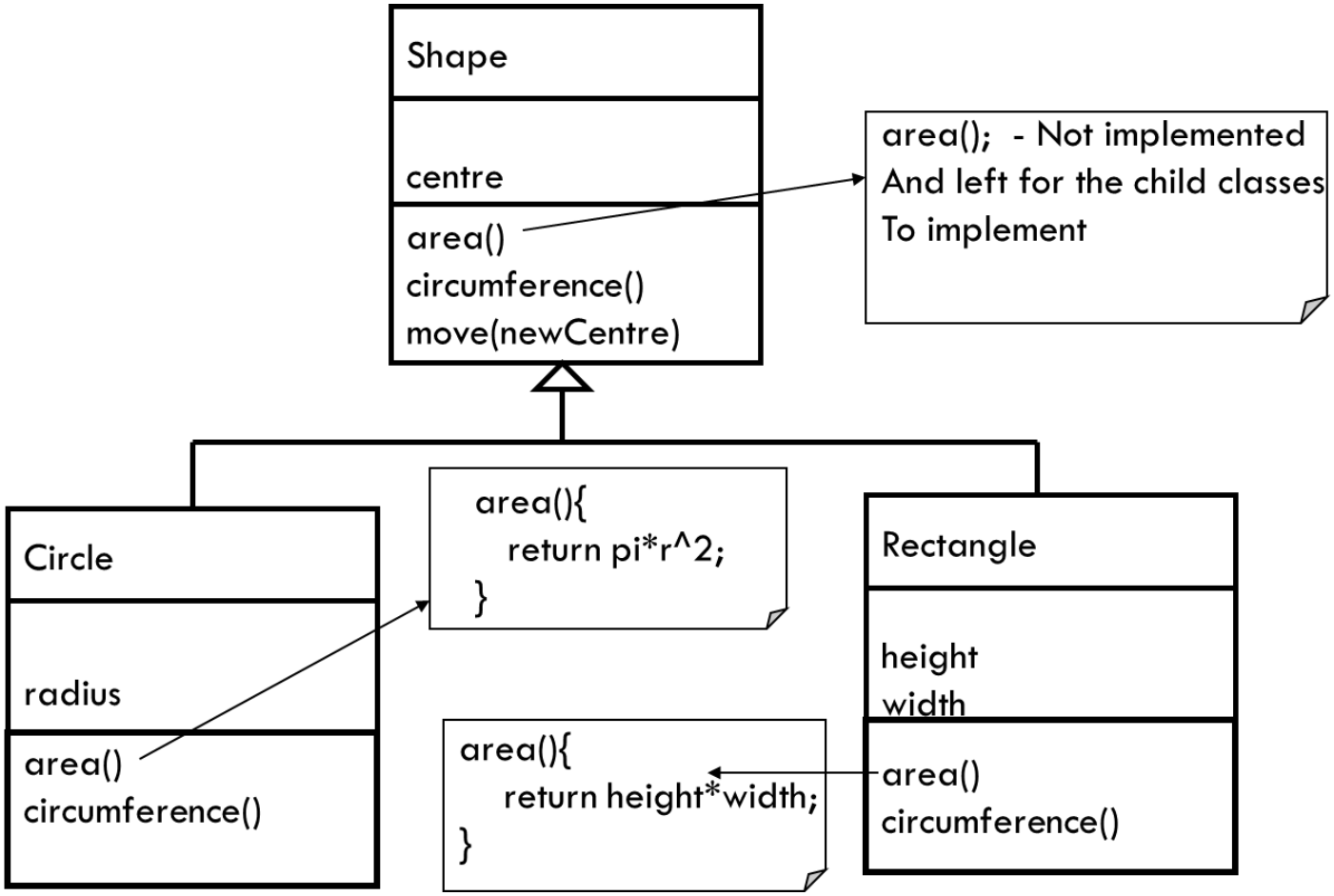


Uses of Inheritance
Reuse

- If multiple classes have common attributes/methods, these methods can be moved to a common class parent class.
- This allows reuse since the implementation is not repeated.
- Example:
  - Rectangle and Circle method have a common method move(), which requires changing the centre coordinate.



Uses of Inheritance
Specialization

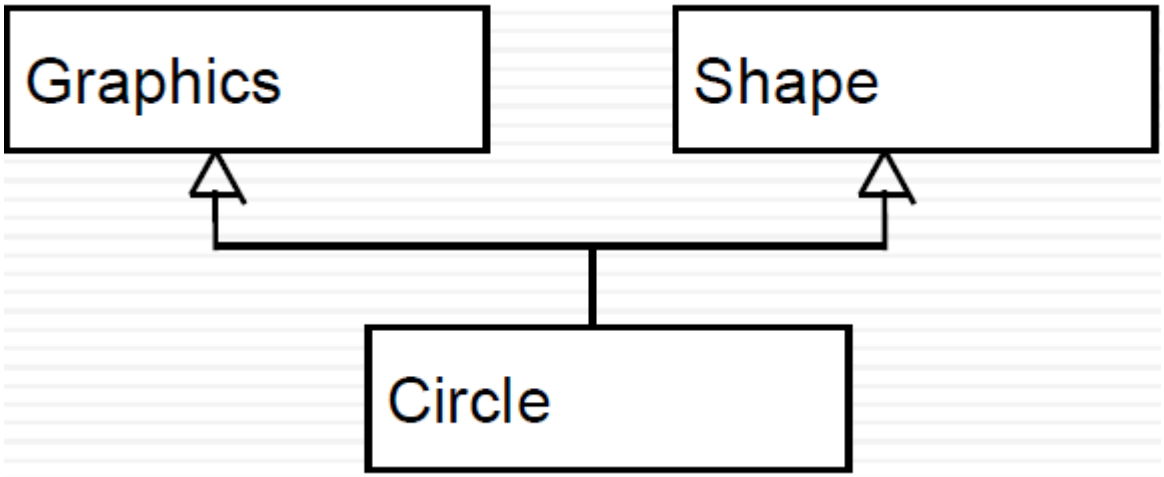


Uses of Inheritance
Common Interface and Extnesion

- All the operations that are supported for Rectangle and Circle are the same.
- Some methods have common implementation and others don't.
  - **move**() operation is common to classes and can be implemented in parent.
  - **circumference**(), **area**() operations are significantly different and have to be implemented in the respective classes.
- The Shape class provides a **common interface** where all 3 operations move(), circumference() and area().
- **Extend** functionality of a class.
- Child class adds new operations to the parent class but does not change the inherited behavior.
  - E.g. Rectangle class might have a special operation
  - that may not be meaningful to the Circle class **rotate90degrees**()

Uses of Inheritance
Multiple Inheritance

- Inherit properties from **more** than one class.
- This is called **Multiple** Inheritance
- Not in Java!
  - Why?



Polymorphism
Object Overloading

- Polymorphic which means “many forms” has Greek roots.
  - Poly → many
  - Morphos → forms.
- In OO paradigm polymorphism has many forms.
- Allow a single **object**, **method**, **operator** associated with different meaning depending on the type of data passed to it.
- An object of type Circle or Rectangle can be assigned to a Shape object. The behavior of the object will depend on the object passed.

```
circleA = new Circle();      Create a new circle object

Shape  shape = circleA;
shape.area();   area() method for circle class will be executed


rectangleA = new Rectangle(); Create a new rectangle object
shape= rectangle;
shape.area()     area() method for rectangle will be executed.
```

Polymorphism
Method Overloading

- Multiple methods can be defined with the same name, **different** input arguments.
  - Method 1 → initialize(int a)
  - Method 2 → initialize(int a, int b)
- Appropriate method will be called based on the input arguments.
  - initialize(2) → Method 1 will be called.
  - initialize(2,4) → Method 2 will be called.

Polymorphism
Operator Overloading

- Allows regular operators such as +, --, *, / to have different meanings based on the type.
  - E.g. + operator for Circle can re defined
    - Circle c = c + 2;
    - Stack s = s + ‘a’
- **Not** supported in JAVA. C++ supports it.
  - It is confusing
  - Think of overloading the = sign …

Static vs. Dynamic Binding

- Binding is what happens when a method invocation is bound to an implementation
  - Involves lookup of the method in the class, or one or its parents
  - Both method names and parameters are checked
- Binding can happen at two different times
  - Compile time == static binding
  - Run time == dynamic binding
- Overloading Vs Overriding

Overloading vs. Overriding

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

Basic differences between C++ and JAVA

Language Examples: C++

- Based on C **struct** type and Simula 67 classes
- The **class** is the encapsulation device
- All of the class instances of a class share a **single** copy of the member functions
- Each instance of a class has its own **copy** of the class data members
- Class instances can be either **stack** dynamic or **heap** dynamic
- Heap dynamic classes are created with **new** and destroyed by **delete**
- Information Hiding
  - **Private** clause for hidden entities
  - **Public** clause for interface entities
  - **Protected** clause for inheritance

Language Examples: C++

- Constructors
  - Functions to initialize the data members of
  - May also allocate storage if part of the object is heap-dynamic
  - Can include parameters to provide parameterization of the objects
  - Implicitly called when an instance is created
  - Can be explicitly called
  - Name is the same as the class name
- Destructors
  - Functions to cleanup after an instance is destroyed; usually just to reclaim heap storage
  - Implicitly called when the object's lifetime ends
  - Can be explicitly called
  - Name is the class name, preceded by a tilde (~)

An Example in C++

```
class stack {
        private:
                int*stackPtr, maxLen, topPtr;
        public:
                stack() { // a constructor
                        stackPtr= new int[100];
                        maxLen= 99;
                        topPtr= -1;
                };
                ~stack () {delete [] stackPtr;};
                void push (intnum) {…};
                void pop () {…};
                int top () {…};
                int empty () {…};
}
```

Language Examples: Java

- Similar to C++, except:
  - All user defined types are **classes** (no structs)
  - All objects are allocated from the **heap** and accessed through reference variables
  - Individual entities in classes have access control modifiers (**private** or **public**), rather than clauses
  - Java has a second scoping mechanism, **package** scope, which can be used in place of friends
    - All entities in all classes in a package that do not have access control modifiers are visible throughout the package
  - **No destructor**: since we have the garbage collection mechanism

Differences between Java and C++

- Global variables
  - It is impossible to create a global variable that is outside of all classes. The only way to create a global variable is by declaring public static variable in a class.
- GOTO
  - Java has no goto statement.
- Pointers
  - Java has no way to manipulate pointers (mem addresses) directly. You cannot convert an integer to a pointer, you cannot dereference an arbitrary memory address.
- Memory Allocation
  - Java has no malloc or free() functions.
- Data types
  - Hardware dependent data types such as int and char* lead to non portable code. Java uses a standard size irrespective of the H/W.
- No header files.
- No preprocessor.

Access Protection Modifiers in Java

class \ have access to	Private Elements	Default Elements	Protected Elements	Public Elements
own class (Base)	Yes	Yes	Yes	Yes
subclass - same package (SubA)	No	Yes	Yes	Yes
class - same package (AnotherA)	No	Yes	Yes	Yes
subclass - another package (SubB)	No	No	Yes/No	Yes
class - another package (AnotherB)	No	No	No	Yes

http://www.artima.com/objectsandjava/webuscript/PackagesAccess1.html