1.1

Object Oriented Programming

Jean-Charles Régin

Master 1 Informatique – 2024

List of integers

- How can we design a list of integer in Java

Example: lists (code Java)


```
class Node {
    Node p_next ;
    int data ;
};

class List {
    List(){…}
    Node first ;
    Node last ;
};
```

Example: lists (code Java)


```
class Node {
    Node p_next;
    int data;
    Node( int index){data=index;}
    void print(){…}
}
boolean check_item( List list, Node node ) {
    boolean found = list.owns_item( node );
    if( found ) {
        node.print();
    }
    return found;
}
// better: do not show the internal data structure
boolean check_item( List list, int data ) {
    boolean found = list.owns_item( data );
    if( found ) {
        node.print();
    }
```

How to generalize?

- We would like to have data (any type) instead of int.
- Solution 1:
  - Associate an int with each data
  - How to get the data from the int ?
    How to get the int from a data ?
    - Array of Data, add int to data
    - Hash Table
- Solution 2:
  - Add a pointer to void* (Object in java)
    - Need downcasting
  - Add a pointer to an object with functions

Question

- Write a function which prints all the elements of a list
  - How to call print(data) ou data.print() ?

Example: lists (code Java)


```
class Node {
    Node p_next;
    Data data;
    Node(Data d, Node next){
          data=d;
          p_next=next;}
    void print(){data.print();}
}
class List {
    List(){first=last=null;}
    Node first ;
    Node last ;
    void add(Data d){
          first=new Node(d,first);
          if (last==null){last=first;}
    }
    void print(){
          Node iter=first;
       while(iter != null){
              iter.print();
           iter=iter.p_next;
          }
    }
};
```

Et Data ?

Example: lists (code Java)


```
interface Data {
    void print();
}
class Student implements Data {
    int num;
    void print(){
        System.out.println(num + " ");
    }
};
```

Objects

- An object is an encapsulation of both functions and data
- **Objects are an Abstraction**
  - represent real world entities
  - Classes are data types that define shared common properties or attributes
  - Objects are instances of a class
- **Objects have State**
  - have a value at a particular time
- **Objects have Operations**
  - associated set of operations called methods that describe how to carry out operations
- **Objects have Messages**
  - request an object to carry out one of its operations by sending it a message
  - messages are the means by which we exchange data between objects

OO Perspective

- Let's look at the Rectangle through object oriented eyes:
- Define a new type Rectangle (a class)
  - Data
    - width, length
  - Function
    - area()
- Create an instance of the class (an object)
- Request the object for its area

- Define a class that encapsulates the knowledge necessary to answer the question - here, what is the area of the rectangle.

Example Object Oriented Code


```
class Rectangle{
private:
    int width, length;
public:
    Rectangle(int w, int l){
                width = w;
                length = l;
          }
    int area(){
          return width*length;
    }
};
```

```
main(){
    Rectangle rect(3, 5);
    cout << rect.area()<<endl;
}
```

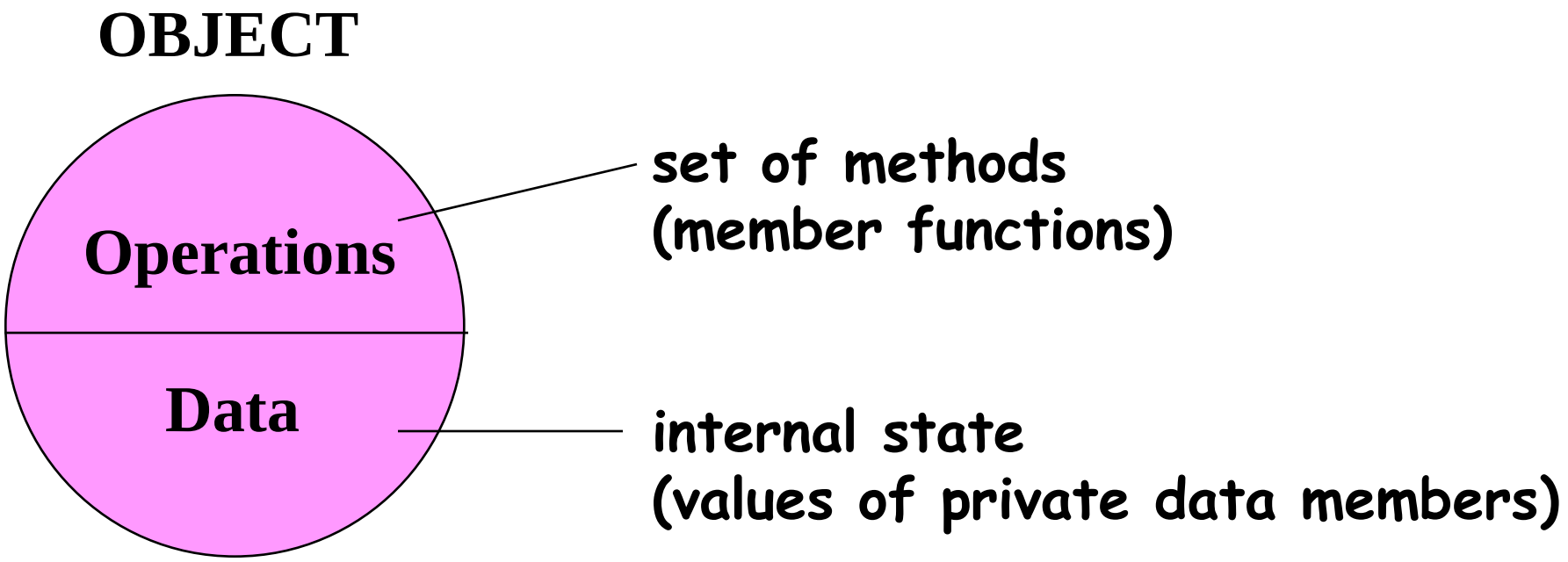
Object-Oriented Programming Languages

- Characteristics of OOPL:
  - Encapsulation
  - Inheritance
  - Polymorphism
- OOPLs support :
  - Modular Programming
  - Ease of Development
  - Maintainability

Characteristics of OOPL

- **Encapsulation:** Combining data structure with actions
  - Data structure: represents the properties, the state, or characteristics of objects
  - Actions: permissible behaviors that are controlled through the member functions
  - Data hiding: Process of making certain data inaccessible
- **Inheritance:** Ability to derive new objects from old ones
  - permits objects of a more specific class to inherit the properties (data) and behaviors (functions) of a more general/base class
  - ability to define a hierarchical relationship between objects
- **Polymorphism:** Ability for different objects to interpret functions differently

What is an object?



Declaration of an Object

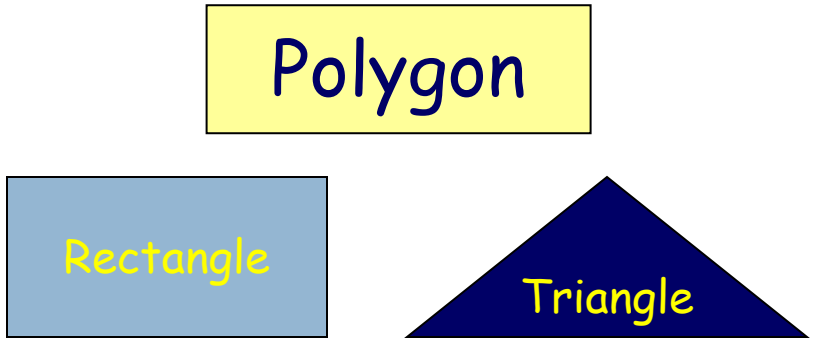

```
class Rectangle
{
    private:
        int width;
        int length;
    public:
        void set(int w, int l);
        int area();
};
```

```
main()
{
    Rectangle r1;
    Rectangle r2;

    r1.set(5, 8);
    cout<<r1.area()<<endl;

    r2.set(8,10);
    cout<<r2.area()<<endl;
}
```

Inheritance Concept

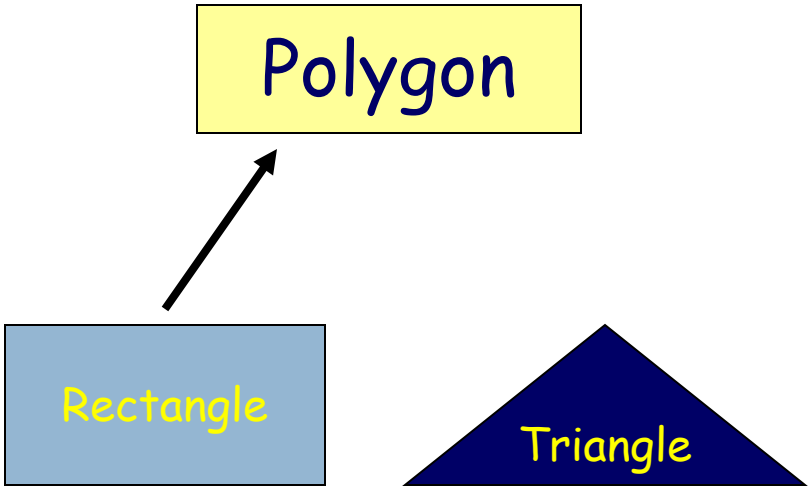



```
class Polygon{
  private:
    int numVertices;
    float *xCoord, *yCoord;
  public:
    void set(float *x, float *y, int nV);
};
```

```
class Rectangle{
  private:
    int numVertices;
    float *xCoord, *yCoord;
  public:
    void set(float *x, float *y, int nV);
    float area();
};
```

```
class Triangle{
  private:
    int numVertices;
    float *xCoord, *yCoord;
  public:
    void set(float *x, float *y, int nV);
    float area();
};
```

Inheritance Concept



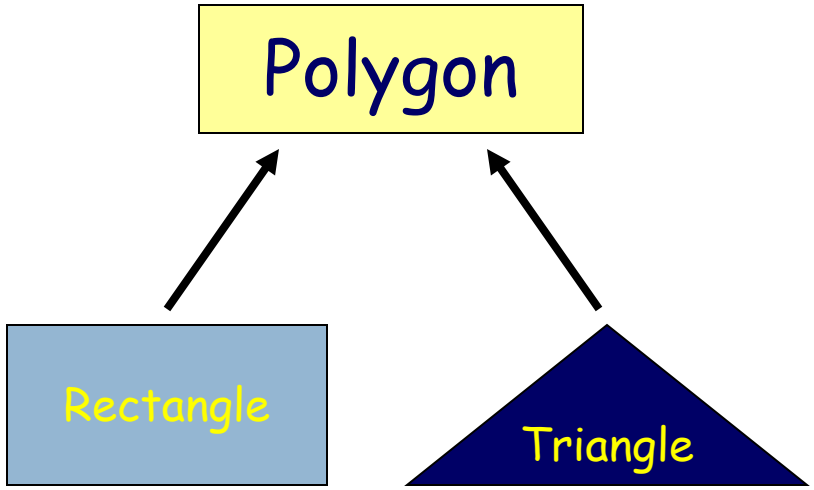

```
class Polygon{
    protected:
      int numVertices;
      float *xCoord, float *yCoord;
    public:
      void set(float *x, float *y, int nV);
};
```

```
class Rectangle : public Polygon{
    public:
      float area();
};
```

⇔

```
class Rectangle{
    protected:
      int numVertices;
      float *xCoord, float *yCoord;
    public:
      void set(float *x, float *y, int nV);
      float area();
};
```

Inheritance Concept




```
class Polygon{
    protected:
      int numVertices;
      float *xCoord, float *yCoord;
    public:
       void set(float *x, float *y, int nV);
};
```

```
class Triangle : public Polygon{
    public:
     float area();
};
```

⟺

```
class Triangle{
    protected:
      int numVertices;
      float *xCoord, float *yCoord;
    public:
       void set(float *x, float *y, int nV);
       float area();
};
```

Why Inheritance ?

- Inheritance is a mechanism for
  - building class types from existing class types
  - defining new class types to be a
    - specialization
    - augmentation

- Prefer an “is-a” relationship

Define a Class Hierarchy

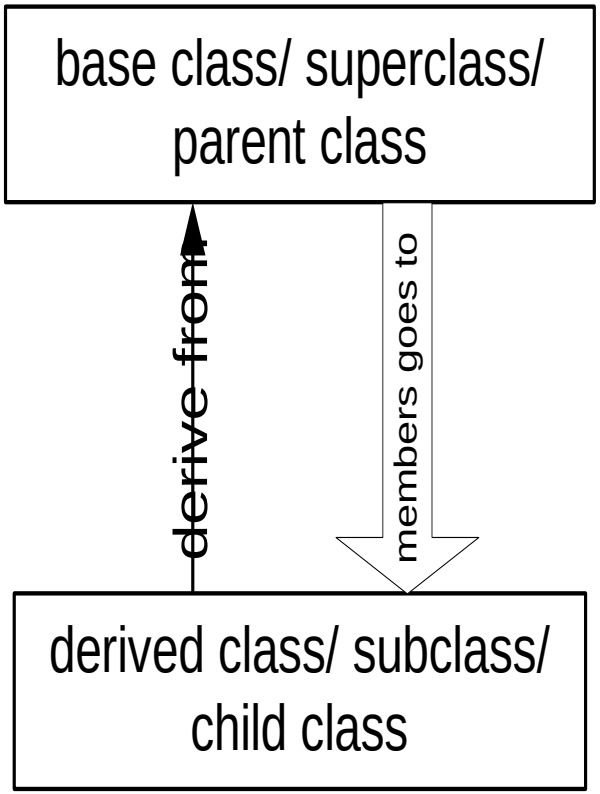
- Syntax:
  - class DerivedClassName : access-level BaseClassName

- where
  - access-level specifies the type of derivation
    - private by default, or
    - public
- Any class can serve as a base class
  - Thus a derived class can also be a base class

What to inherit?

- In principle, every member of a base class is inherited by a derived class
  - just with different access permission

Access Control Over the Members



- Two levels of access control over class members
  - class definition
  - inheritance type

Define its Own Members

- The derived class can also define its own members, in addition to the members inherited from the base class
- A triangle may have a function checking whether it is isoceles

Even more …

- A derived class can override methods defined in its parent class. With overriding,
  - the method in the subclass has the identical signature to the method in the base class.
  - a subclass implements its own version of a base class

```
class A {
  protected:
    int x, y;
  public:
    void print ()
          {cout<<"From A"<<endl;}
};
```

```
class B : public A {
  public:
    void print ()
        {cout<<"From B"<<endl;}
};
```

Exercice

- Write an Array class
- Write a Stack
  - From scratch
  - Derive from Array
    - What's about the set function?
      - Solve the problem
  - Composite from Array

- Which one is the best?

Late binding

- Problem “run time object determination”
- How ?
  - Define a function in the base class.
  - It can be overridden in the derived classes
  - But, a derived class is not required to re-implement the function. If it does not, the base class version is used

Polymorphism Summary:

- When you use unctions, compiler store additional information about the types of object available and created
- Polymorphism is supported at this additional overhead

Abstract Classes & Abstract Functions

- Some classes exist logically but not physically.
- Example : Shape
  - Shape s; // Legal but silly..!! : "Shapeless shape"
  - Shape makes sense only as a base of some classes derived from it. Serves as a "category"
  - Hence instantiation of such a class must be prevented

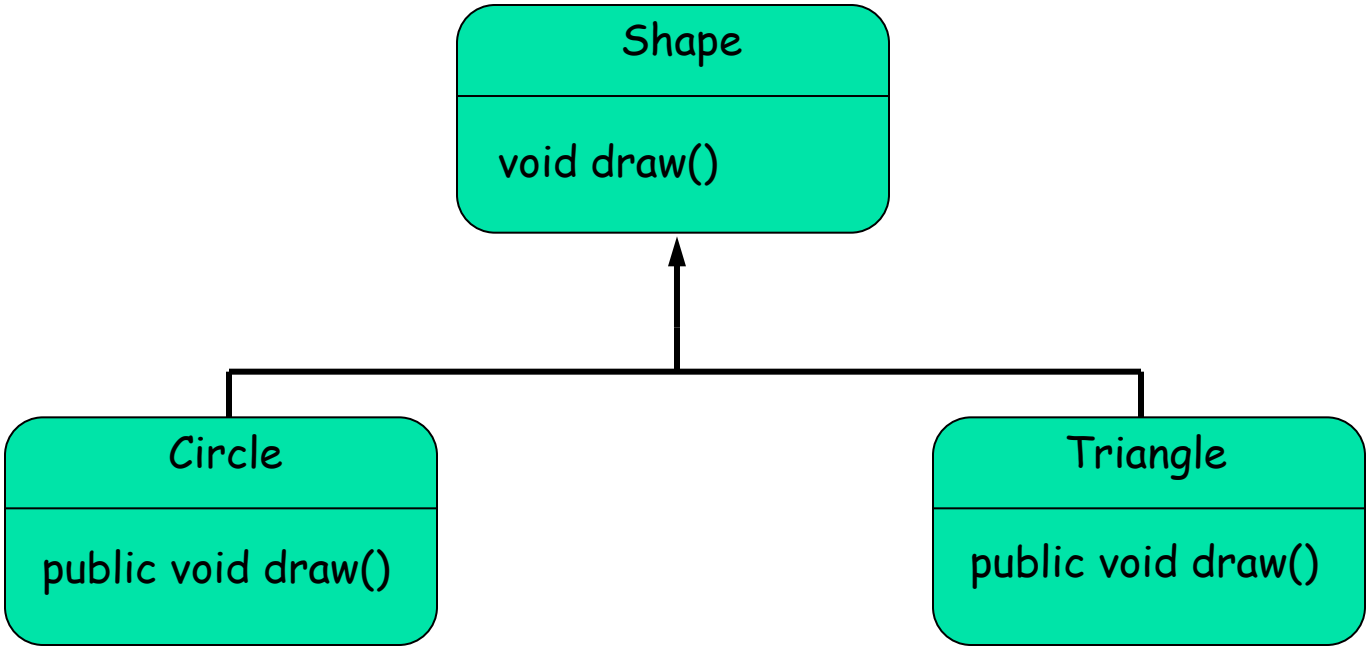
```
class abstract Shape
//Abstract
{
  //Abstract Function
  abstract void draw();
}
```

A class with one or more pure virtual functions is an Abstract Class

Objects of abstract class can't be created

```
Shape s; // error : variable of an abstract class
```

Example


Shape
void draw()
Circle
public void draw()
Triangle
public void draw()

Abstract function

- An abstract function not defined in the derived class remains an abstract function.
- Hence derived class also becomes abstract

```
class Circle extends Shape { //No draw() - Abstract
   public void print(){
     print("I am a circle");
   }
class Rectangle extends Shape {
   public void draw(){ // Override Shape::draw()
     print("Drawing Rectangle");
   }
```

Abstract functions : Summary

- Abstract functions are useful because they make explicit the abstractness of a class
- Tell both the user and the compiler how it was intended to be used
- Note : It is a good idea to keep the common code as close as possible to the root of you hierarchy

Exercice

- Consider the code you wrote for the list. Define an Employee as equivalent of a student. Then find a way to have a list mixing Student and Employee.

Exercice

- You are given a Class with property defined at the creation. The property is either true or false
- The following code is given in this class
  - void run(){if (property) f() else g();}
- Propose a new hierachy of classes such that there is no longer an if in the run function

Exercice

- Same for a logger with different levels
  - ALL All levels including custom levels.
  - DEBUG Designates fine-grained informational events that are most useful to debug an application.
  - INFO Designates informational messages that highlight the progress of the application at coarse-grained level.
  - WARN Designates potentially harmful situations.
  - ERROR Designates error events that might still allow the application to continue running.
  - FATAL Designates very severe error events that will presumably lead the application to abort.
  - OFF The highest possible rank and is intended to turn off logging.
  - TRACE Designates finer-grained informational events than the DEBUG.