An Introduction to Object Orientation

Rushikesh K Joshi

Indian Institute of Technology Bombay

rkj@cse.iitb.ac.in

A talk given at Islampur

Abstractions in Programming

- Control Abstractions
  - Functions, function calls, recursion
  - Assignment statement
  - Sequential execution
  - If then else, while, repeat, case, for statements
  - Threads
  - Coroutines
  - Contiuations and mobility
  - Rules and inference
- Control abstractions can control data flows

Abstractions in Programming

- Data Abstractions
  - symbols and lists
  - Types: int, bool, char, float..
  - Structures
  - Unions, enumerated types
  - Arrays, Vectors
- operations supported on data abstractions are mostly general: read, write

Towards Richer Abstractions

- The above control and data abstraction are low level
- High level abstractions need to be composites of these
  - Besides function composition, structures:
  - it makes sense to combine data and control together to form an interesting composite abstraction

Examples of Richer Abstraction

- File at OS level
  - Data: stream of bytes
  - Operations supported: open, close, read, write, rewind, seek
- Process at OS level
  - Data: control and data segments, page tables, open files, priority..
  - Control: create, terminate, suspend, resume, trace
- Stack Data structure
  - Data: elements arranged in the form of stack
  - Control: create, delete, push, pop, top
- Table in a spreadsheet/GUI
  - Data: rows, columns, content
  - Operations (control): create, delete, add/del row/column, insert element
- Name server
  - Data: name-location bindings arranged in a hierarchy
  - Operations: add new binding, delete existing binding, create/delete namespaces

Compare These with Some Examples of Abstraction in Real life

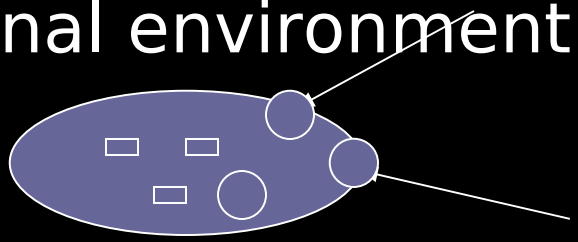
- Fan
  - Data: motor, capacitor ..
  - Operations: switch on, off, set speed
- Tape
  - Data: internal circuits, cassette holder
  - Operations: switch on/of, open/close cassette holder, play, rewind, forward, record, pause, continue
  - It’s a composite object: player/recorder + cassette holder
- Washing Machine, car, scooter, TV set, mixer…

They have something in common: Towards Object Abstractions

- It is convenient to think of abstractions in terms of the data that they possess along with the operations which they allow on them
  - Data: Internal
  - Operations: Expose for External Use
- User only worries about how to use an abstraction but now how it is implemented
- Such simplicity at high level is possible due to
  - Thinking data and high level control together
  - Separating data from exposable operations on them
  - Hiding data from external environment

Two Basic Principles of Object Orientation

- Abstraction
  - Object abstraction: data + observable behavior
- Encapsulation
  - Only observable behavior is exposed, the rest (mainly the data) is hidden from external environment



Exercise

- Define following objects in terms of their observable behavior
  - Stack
  - List
  - Account
  - Button
  - Transaction
  - Semaphore

Object Orientated Programming Languages

- Provide a core abstraction for defining such objects
  - Class and instances: class based languages
  - Only instances: prototyped based languages
- In addition to the core object abstraction, the benefits of object orientations are reaped through two additional principles of
  - *inheritance* and
  - *polymorphism through inheritance*

A Class and its implementation

```
Class X {
 int x;
 public:
   int add(int p);
   int subtract(int p);
 };
Int X:: add (int p) {x=x+p;};
Int X::subtract (int p) {x=x-p;};
```

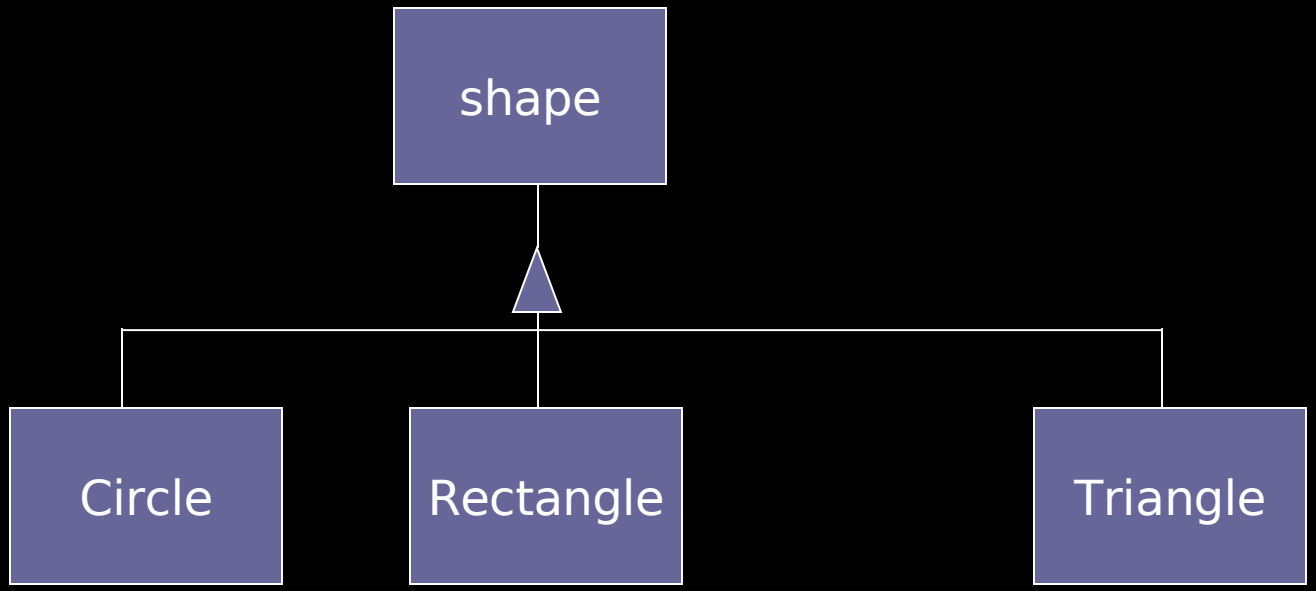
Another Example

- class Complex {
- private:
- int i ; // real component
- int j ; // imaginary component
- public:
- Complex (int x, int y) { i=x; j=y; }
- void add (Complex a) ;
- void printState (void);
- };
- void Complex::add(Complex) { i += a.i ; j += a.j; }
- Void Complex :: printState (void) { cout << i << " + j" << j << "\n" ; }

Inheritance

- Mechanism for
  - Pure Extension
    - C1={f,g,h}
    - C2=C1+{p,q}
  - Specialization
    - C1={f,g,h}
    - C2=C1 with f’ to be treated as f, rest of o1 as it is+{p,1}
  - Polymorphism
    - Use instances of C2 where instances of C1 are required

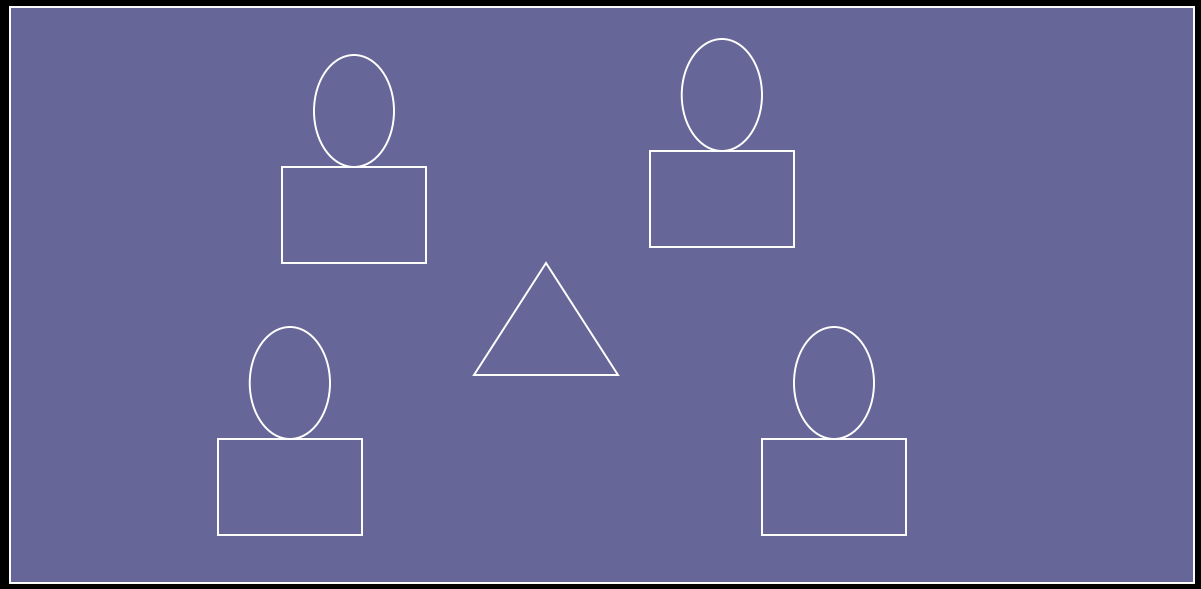
An Example of Inheritance Hierarchy



Exercise

- Implement classes shape, circle, rectangle and triangle to support following abilities for all shapes:
  - Create, delete, move, clone
- What do you keep in the superclass?
  - For use at it is
  - For specialization and subsequent polymorphism

An Application that uses this hierarchy: A Graph Drawing Editor



Benefits of inheritance

- Superclass Shape contains most common properties
- It also contains abstract member functions which are applicable to all shapes
- Abstract member functions are concretely defined in subclasses
- Application has a lot of code written in terms of superclass shape

Using Polymorphism

- Mouselistner (event e, shape s) {
- if (e==drag)
  - s→moveTo (currentX, currentY);

  }

The above code is applicable to all types of shapes.

In absence of polymorphism, a switch statement had to be used

Dynamic Binding of method dispatches results in Polymorphism

- Mouselistner (event e, shape s) {
- if (e==drag)
  - s→moveTo (currentX, currentY);

  }

The moveTo method bound at runtime

S is supertype (static type), but actual object's type (dynamic type) determines which member function should be dispatched

Abstract Superclasses

- Meant for specialization only
- Not for instantiation directly
- Represent most common behavior for all its subclasses
  - E.g. class shape in above example
- All methods are unimplemented (pure virtual in C++/Java)

Hooks, Template Methods and Concrete Methods

```
Class X {
 public:
   f()=0;     //hook
   g() {.....; f(); ....;}     // template method
   h () {....} .// concrete method
};
```

Frameworks/Design Patterns use these three meta-patterns extensively

Terminologies

- Class
- Instance/Object
- Implementation
- Interface Information/Data hiding
- Encapsulation
- Inheritance
- Superclass/Base class
- Subclass/Derived Class

Contracts

- Between class and itself
  - Can see all its data and member function
- Between class and its external environment
  - Environment sees only public member functions/public data
- Between class and its subclasses
  - Subclasses get to see protected members

Purity of Object Orientation

- C++
  - Supports object oriented but does not enforcing
  - Functions which are non-member functions are acceptable
  - Encapsulation can be broken
- Java
  - Enforces classification
    - Even main is a member function
- Eiffel
  - Design on the basis of contracts
- Smalltalk
  - Even control constructs are object oriented
  - Classes are also instances

Tree Vs. Forest

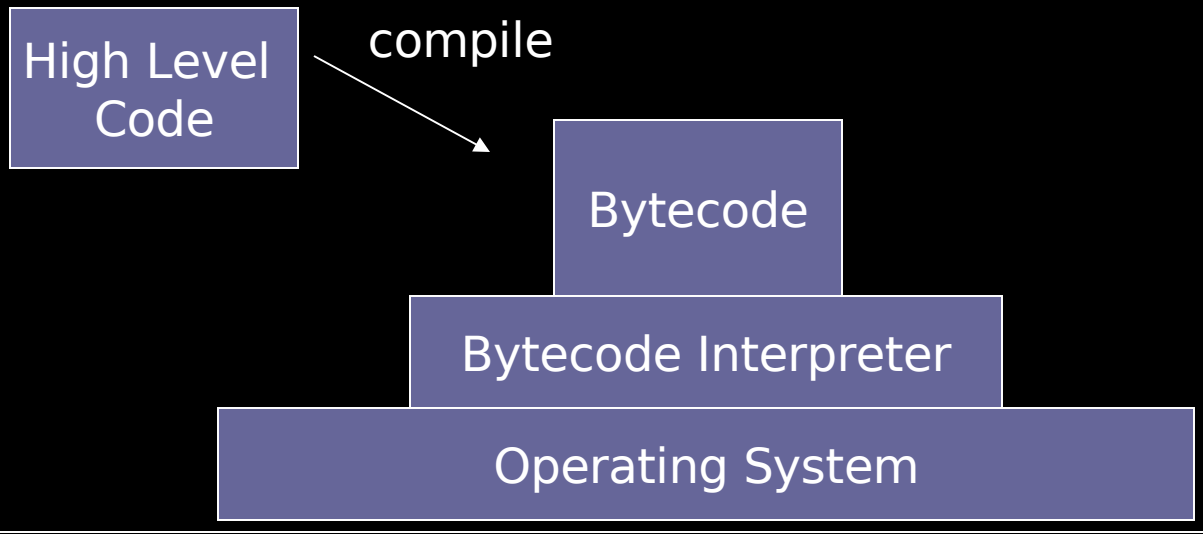
- Most common superclass for all objects in the language
  - Class Object
    - In smalltalk, Java
- Class hierarchies are not implicitly linked
  - As in C++

Template classes Vs. Using class Object

- For writing generic code
- A generic code is applicable to different types
  - C++ Employees template classes
  - Java relies on super-most generic Class Object

Bytecodes for Portability of Programs

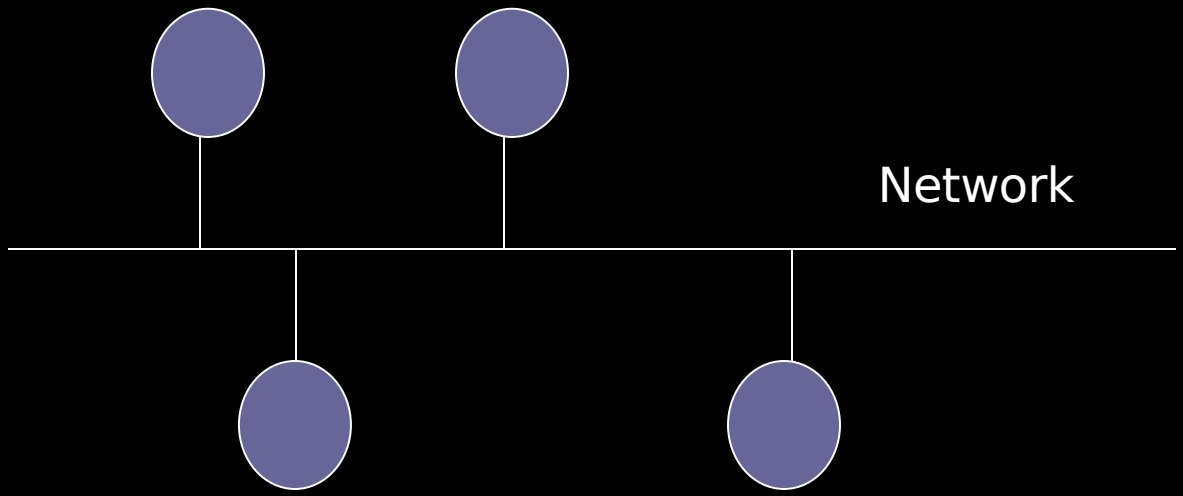
- Intermediate language
- Bytecode interpreter is made available for a specific OS
- Used in interpreter based OO languages such as Smalltalk and Java



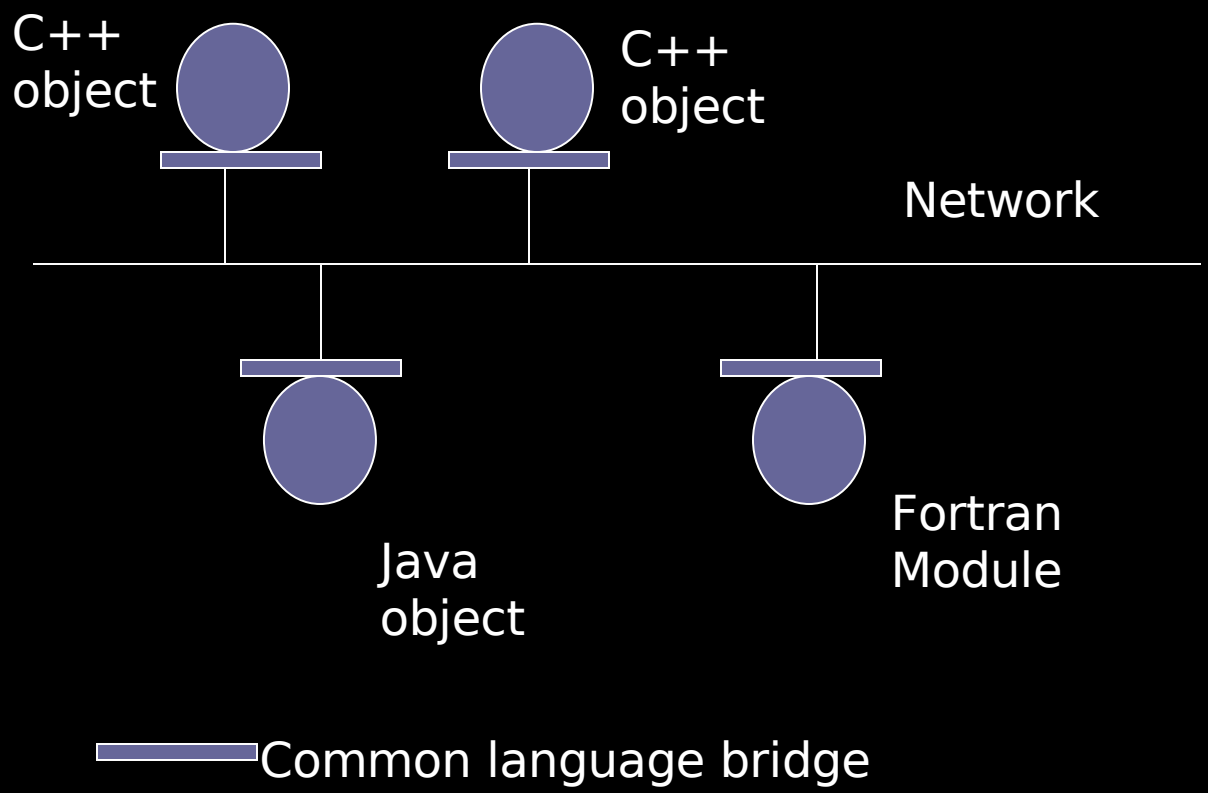
Core Language and Libraries

- Core language contains set of keywords and its control, data and object abstractions
- Some languages also supports abilities such as treads and interprocess communication
- Most of the application development relies on the library/package support that the language development environments support
  - Gui, database connectivity, distribution and component orientation, strings, collections, patterns, i/o etc.

Distributed Objects



Interoperability



Design Patterns

- Commonly occurring non-trivial designs
- Collection of classes connected in a specific configuration
- Creational patterns
  - Singleton, factory, prototype
- Structural patterns
  - Proxy, adapter
- Behavioral patterns
  - Strategy, visitor, observer, chain of responsibility

Object Oriented Analysis and Design Methods

- How to capture requirements?
  - Use cases
- How to identify objects?
  - Coad and Yourdon method
  - CRC of Kent beck
- How to represent designs?
  - Static behaviors
    - Class diagrams
  - Dynamic behavior
    - Interaction diagrams, state diagrams, activity diagrams, collaboration diagrams
  - Deployment/Packaging Behavior
    - Package diagrams
    - Deployment diagrams (machines and component allocations)
- Modeling Languages: e.g. UML

Some Recent Developments

- Software Architectures
  - Architectural description languages
  - Architecture to realization mappings
  - Model Driven Architectures
- Component Technologies and Web Services
- Aspect Orientation and Advanced separation of concerns