Object-Oriented Programming

Programming with Data Types
to enhance *reliability* and *productivity*
(through reuse and by facilitating evolution)

- Object (instance)
  - State (fields)
  - Behavior (methods)
  - Identity
- Class
  - code describing implementation of an object

- Data Abstraction
- Modularity
- Encapsulation
- Inheritance
- Polymorphism

Abstraction

- *General:* Focus on the meaning
  - Suppress irrelevant "implementation" details
- *Programming Languages :*
  - Assign *names* to recurring patterns
    - Value : *constant identifier*
    - Expression : *function*
    - Statements : *procedure*
    - Control : *loop, switch*
    - Value/ops : *interface*

Data Abstraction

- Focus on the meaning of the operations (*behavior*), to avoid over-specification.
- The representation details are confined to only a small set of procedures that create and manipulate data, and all other access is *indirectly* via only these procedures.
  - Facilitates code evolution.

Data Abstraction : Motivation

- Client/user perspective *(Representation Independence)*
  - Interested in ***what*** a program does, not *how*.
  - Minimize irrelevant details for clarity.
- Server/implementer perspective *(Information Hiding)*
  - Restrict users from making *unwarranted assumptions* about the implementation.
  - Reserve right to change representation to improve performance, … (*maintaining behavior*).

Data Abstraction : Examples

- Queues (empty, enQueue, deQueue, isEmpty)
  - array-based implementation
  - linked-list based implementation
- Tables (empty, insert, lookUp, delete, isEmpty)
  - Sorted array (logarithmic search)
  - Hash-tables (*ideal*: constant time search)
  - AVL trees (height-balanced)
  - B-Trees (optimized for secondary storage)

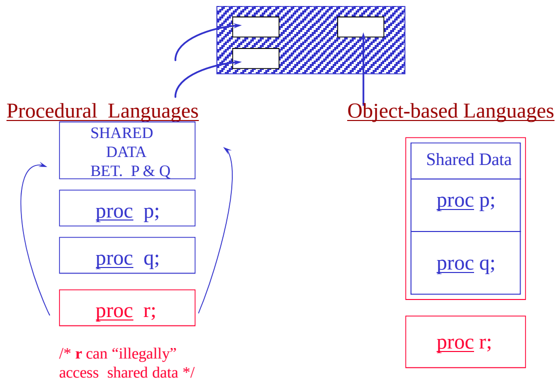
Modularity

- Aspect of syntactically grouping related declarations. (E.g., fields and methods of a data type.)
  - Package/class in Java**.**
- In OOPLs, a ***class*** serves as the basic unit for decomposition and modification. It can be *separately compiled*.

Criteria for Modular Design

- Supports decomposition for division of labor
- Supports composition for reuse
- Supports continuity (incremental updates) for extendibility and smoother evolution
- Supports understandability
- Supports protection and isolation

ENCAPSULATION



Encapsulation

- Controlling visibility of names.
- Enables *enforcing* data abstraction
  - Conventions are no substitute for enforced constraints.
- Enables mechanical detection of typos that manifest as "illegal" accesses. (Cf. problem with global variables)

Data Abstraction : *Summary*

- *Theory* : Abstract data types
- *Practice* :
  - Information hiding ("server")
  - Representation independence ("client")
- *Language* :
  - Modularity
  - Encapsulation

Inheritance : Subclasses

- Code reuse
  - derive *Colored-Window* from *Window* (also adds fields/methods)
- Specialization: Customization
  - derive *bounded-stack* from *stack* (by overriding/redefining *push*)
- Generalization: Factoring Commonality
  - code sharing to minimize duplication
  - update consistency

Inheritance/Redefinition : *Example*

```
import java.awt.Color;
class Rectangle {
    int  w, h;
    Rectangle (int ww, int hh) {
            w = ww;       h = hh;
    }
    int  perimeter () {
          return ( 2*(w + h) );
    }
}
```



```
class  ColoredRectangle extends Rectangle {
  Color c;                         // inheritance
  ColoredRectangle (Color cc, int w, int h) {
            super(w,h);     c = cc; }
}

class  Square extends Rectangle {
  Square(int w) {
          super(w,w);  }
  int  perimeter () {          // overriding
          return ( 4*w );   }
}
```


Open-closed principle

- A class is *closed* because it can be compiled, stored in a library, and made available for use by its clients.
  - **Stability**
- A class is *open* because it can be extended by adding new features (operations/fields), *or* by redefining inherited features.
  - **Change**

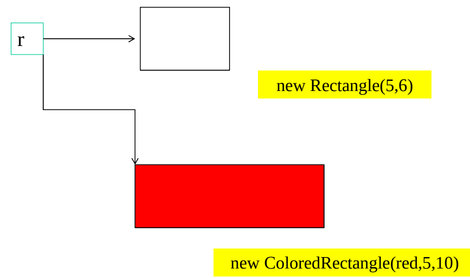
Polymorphism (*many forms*)

- Integrating objects that exhibit a common behavior and share code.
- Unifying heterogeneous data.
  - E.g., *moving*, *resizing*, *minimizing*, *closing*, etc windows and colored windows

Polymorphism : *Example*

```
class Eg {
   public static void main (String[] args) {
      Rectangle  r  =  new Rectangle(5,6);
      System.out.println( r.perimeter() );
      r  = new ColoredRectangle(Color.red,5,10) ;
      System.out.println( r.perimeter() );
   }
}
```


Polymorphic Variable r



Signature

- ***Signature*** of a procedure is the sequence of types of formal parameters and the result of a function.
- ***Signature*** of a function also includes its return type.
  - + : *real* x *real* –> *real*
  - push : *int* x *stack* –> *stack*
  - isEmpty : *stack* –> *boolean*
  - 0 : *int*

Overloading

- Same *name* for *conceptually related* but different operations.
  - E.g., print(5); print("abc"); print(Table);
  - E.g., 1 + 2, "abc" + "…" + "xyz"
- Ambiguity resolved on the basis of contextual information (*signature*)
  - Scalar Multiplication:
    - 2 * [1,2,3] = [2,4,6]
  - Dot-product:
    - [1,2] ***** [1,2] = 5
  - Cross-product:
    - [1,2,0] * [1,2,0] = [0, 0, 0]

Binding

- Associating a method call with the method code to run
  - Resolving ambiguity in the context of overloaded methods
  - Choices for binding time
    - *Static*: Compile-time
    - *Dynamic* : Run-time

Binding: Static *vs* Dynamic

- Static binding (resolved at compile-time)
    - Vector . **mul**(Number)
    - Vector . **mul**(Vector)
  - *both* **mul** *defined in one class*
- Dynamic binding (resolved at run-time)
    - (Array) Stack . **push**(5)
    - (List) Stack . **push**(5)
  - the two **push***es* defined in different classes

Polymorphism and Dynamic Binding

- Integrating objects that share the same behavior/interface but are implemented differently.
    - **Representation independence of clients. (Sharing/Reuse of "high-level" code.)**
  - E.g., *search*ing for an identifier in an *array of tables*, where each *table* can potentially have a different implementation.
  - E.g., *push*ing a value on a *stack* or a *bounded stack*.

Dynamic Binding : *Example*

```
class Eg {
   public static void main (String[] args) {
      Rectangle []  rs = { new Rectangle(5,6),
        new  ColoredRectangle(Color.red,1,1),
        new  Square(3)} ;
        for   (int i = 0 ; i < rs.length ; i++ )
          System.out.println( rs[i].perimeter() );
}}
```

Polymorphic data-structure (rs)

Polymorphic variable (rs[i])

Dynamic Binding (perimeter())

Rendition in C++

```
#include <iostream>
using namespace std;
class Rectangle {
    protected:
            int  w, h;
    public:
            Rectangle (int ww, int hh) {
                w = ww;
                h = hh;
            }
            virtual
            int  perimeter () {
                return ( 2*(w + h) );
            }
};
```



```
class  ColoredRectangle : public Rectangle {
   private:                          // inheritance
        int c;
   public:
        ColoredRectangle (int cc, int w, int h) :
                Rectangle(w,h) {
                        c = cc;
        }
};
class  Square : public Rectangle {
   public:
        Square(int w) : Rectangle(w,w) {}
        int  perimeter () {            // overriding
                return ( 4*w );        // protected, not private
        }
};
```



```
void main (char* argv, int argc) {
        Rectangle  r (5,6);
        cout  <<  r.perimeter()   << endl;
        ColoredRectangle cr (0,1,1) ;
        r = cr;             // coercion (truncation)
        cout  << r.perimeter()   << endl
              << cr.perimeter()  << endl;  // inheritance
        Square s = Square(5);
        r = s;              // NOT polymorphism
        cout    << r.perimeter()  << endl;
        cout   << s.perimeter()   << endl; // static binding
}
```



```
void main (char* argv, int argc) {
        Rectangle*  r  =  new Rectangle(5,6);
        cout <<   r–>perimeter()   << endl;
        r  = new ColoredRectangle(0,1,1) ;
        cout   << r–>perimeter()   << endl;
        r  = new Square(5) ;
        cout   << r–>perimeter()    << endl;
            // polymorphism and dynamic binding
            //  perimeter() explicitly declared virtual
}
```

Polymorphic Data Structure and Dynamic Binding in C++

```
void main (char* argv, int argc) {
    const RSLEN = 3;            // coercion, no dynamic binding
    Rectangle rs [RSLEN]= { Rectangle(5,6),
            ColoredRectangle(0,1,1),  Square(5)} ;
    for   (int i = 0 ; i < RSLEN ; i++ )
      cout << rs[i].perimeter() << endl;
}
void main (char* argv, int argc) {
    const RSLEN = 3;            // polymorphism
    Rectangle* rs [RSLEN]= { new Rectangle(5,6),
            new  ColoredRectangle(0,1,1),   new  Square(5)} ;
    for   (int i = 0 ; i < RSLEN ; i++ )
      cout << rs[i]–>perimeter() << endl;
}
```


Summarizing :Java vs C++ vs C#

- Java version uses "references to structures"
  - Employs polymorphism and dynamic binding
- C++ version 1, which resembles Java version, uses "structures"
  - Employs coercion and static binding
- C++ version 2, which differs from Java version on the surface but simulates Java semantics using "references to structures"
  - Employs polymorphism and dynamic binding

Summarizing :Java vs C++ vs C#

As will be seen …

- C# versions combine the syntax of Java and C++ but support only "references to structures" similarly to Java
  - C# version 1 simulates the Java semantics of polymorphism and dynamic binding by overriding when the parent and child method signatures match
  - C# version 2 enables avoiding overriding and dynamic binding due to coincidental signature match

Rendition in C#

```
using System.Drawing;
class Rectangle {
    protected int  w, h;
    public Rectangle (int ww, int hh) {
         w = ww;      h = hh;
    }
    public virtual int perimeter () {
      System.Console.WriteLine( "Rectangle.perimeter() called" );
      return ( 2*(w + h) );
    }
}
class  ColoredRectangle : Rectangle {
    protected Color c;                  // inheritance
    public ColoredRectangle (Color cc, int w, int h):base(w,h) {
         c = cc;
    }
}
```

```
class  Square : Rectangle {
    public Square(int w): base(w,w) {    }
    public override int perimeter () {         // overriding
      System.Console.WriteLine( "Square.perimeter() called" );
      return ( 4*w );
    }
}
class EgA {
   public static void Main (string[] args) {
      Rectangle []  rs = { new Rectangle(5,6),
        new  ColoredRectangle(Color.Red,1,1),
        new  Square(2)
      } ;
      for   (int i = 0 ; i < rs.Length ; i++ )
        System.Console.WriteLine( rs[i].perimeter() );
   }
}
```


Rendition in C#

```
using System.Drawing;

class Rectangle {
    protected int  w, h;
    public Rectangle (int ww, int hh) {
           w = ww;       h = hh;
    }
    public int perimeter () {
       System.Console.WriteLine( "Rectangle.perimeter() called" );
         return ( 2*(w + h) );
    }
}
class  ColoredRectangle : Rectangle {
    protected Color c;                        // inheritance
    public ColoredRectangle (Color cc, int w, int h):base(w,h) {
          c = cc;
    }
}
```



```
class  Square : Rectangle {
    public Square(int w): base(w,w) {    }
    public new int perimeter () {         // unrelated
      System.Console.WriteLine( "Square.perimeter() called" );
      return ( 4*w );
    }
}
class EgA {
   public static void Main (string[] args) {
      Rectangle []  rs = { new Rectangle(5,6),
        new  ColoredRectangle(Color.Red,1,1),
        new  Square(2)
      } ;
      for   (int i = 0 ; i < rs.Length ; i++ )
        System.Console.WriteLine( rs[i].perimeter() );
   }
}
```


Polymorphism and Dynamic Binding : *Examples*

- Viewing various types of files.
  - `*.ps, *.ppt, *.html, *.java`, etc
- Determining values of different kinds of expressions.
  - `variable, plus-expr, conditional-expr`, etc
- Moving/copying file/directory.
- Using same bay (interface) to house CD or floppy drive in a laptop.
  - different device drivers
- Car's "interface" to a driver.

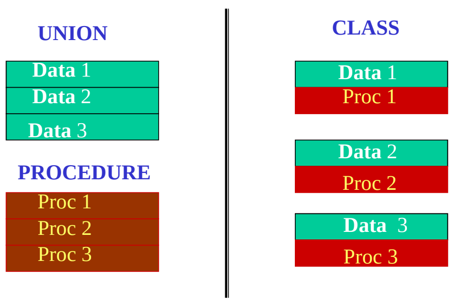
Reuse : *Summary*

- Inheritance and Polymorphism
  - code sharing / reusing implementation
- Polymorphism and dynamic binding
  - behavior sharing / reusing "higher-level" code
  - Accommodating variations in implementation at run-time.
- Generics / Templates
  - Accommodating variations in type

Styles : Procedural *vs* Object-Oriented

• C's *Union type and Switch stmt.* (Pascal's *Variant record and Case stmt.)*	• Java's *extends* for sub-class and *implements* for sub-type.
• Explicit dispatching using switch/case	• Automatic dispatching using type tags
• Addition of new procs. incremental	• Addition of new impl. (data/ops) incremental
• Not ideal from *reuse* and *modularity* view	• Classes in binary form too *extensible*

Styles: Procedural *vs* Object-Oriented



Inter-Class Relationships

"A `CarOwner` *is a* `Person` and *has a* `Car`."

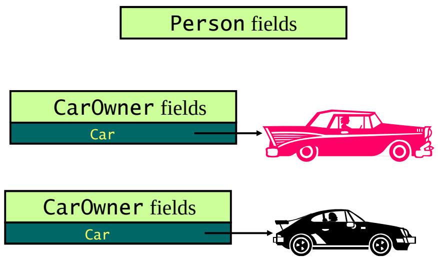
- Composition (*Client Relation*) ("*has a*")
- Inheritance (*Subclass Relation*) ("*is a*")

```
class CarOwner extends Person { Car c; ... }
```

- The difficulty in choosing between the two relations stems from the fact that *when the "is" view is legitimate, one can always take the "has" view instead.*

```
class CarOwner { Car c; Person p; ... }
```

Subclass instance ; Client field



Example : OOP Style *vs* Procedural Style

- **Client**
  - Determine the number of elements in a collection.
- **Suppliers**
  - Collections : Vector, String, List, Set, Array, etc
- **Procedural Style**
  - A client is responsible for invoking appropriate supplier function for determining the size.
- **OOP Style**
  - Suppliers are responsible for conforming to the standard interface required for exporting the size functionality to a client.

Client in Scheme

```
(define (size C)
   (cond
     ( (vector?  C)   (vector-length C) )
     ( (pair?  C)     (length C) )
     ( (string?  C)   (string-length C) )
     (  else          "size not supported") )
))

(size    (vector 1 2 (+ 1 2)))
(size    '(one "two" 3))
```


Suppliers and Client in Java

```
interface  Collection  {  int size(); }

class  myVector  extends  Vector
             implements  Collection {
}
class  myString  extends  String
             implements  Collection {
   public int size() { return  length();}
}
class myArray  implements Collection {
   int[]  array;
   public int size() {return  array.length;}
}

Collection c = new  myVector();  c.size();
```