Object-Oriented Paradigm

1. Objectives

- Object technology helps build complex applications quickly: applications are created from existing components
- Easier design: designer looks at objects as a black box, not concerned with the detail inside
- Software Reuse: Classes are designed so they can be reused in many systems or create modified classes using inheritance
- Enhance the software product quality: Reliability, Productivity, etc
- Support for software adaptability more easily
- Support the move towards distributed systems
- Robustness - classes designed for repeated reuse become stable over time

2. A little Bit of History

- Object-oriented paradigm has its roots back to **Simula** (Simulation of real systems) language that was developed by in 1960 by researchers at the Norwegian Computing Center.
- In 1970, Alan Kay and his research group at Xerox PARK (the Palo Alto Research Center) developed the first pure OOPL: **Smalltalk**.
- In parallel to Alan's work, Bjarne Stroustrup at Bell Laboratories was also developing an extension to C language, called **C++**, that implements the OO concepts.
- By the time of the first major conference on OO programming, in 1986, there were dozens of languages: Object C (1986), Eiffel (1988), Object Pascal, etc.

3. Overview

- The object-oriented paradigm shifts the focus of attention from code to data.

- OO is based on modeling real-world objects

- The general approach of OOP is to view a software system as a collection of interacting entities called ***"objects"*** each of which is defined by an ***identity***, a ***state*** described in terms of ***member variables***, *and* a ***behavior*** described in terms of ***methods*** that can be invoked.

- The basic unit of abstraction is the OBJECT, which encapsulates both state information (in the form of the values of instance variables) and behavior (in the form of methods, which are basically procedures.)

- Most object-oriented programming languages are fairly similar to imperative languages - the paradigm shift is more in how we THINK about problem solving than it is in the style of the programming language we use.

- Examples:
  Simula, Smalltalk, Eiffel, Objective-C, C++, Java

4. Objects

- Philosophical definition: An entity that you can recognize

- In object technology terms: An abstraction of a “real-world” object.
- In software terms: A data structure and associated functions.

- Object technology decomposes a system entirely into objects.

- The object can perform operations: You can ask the object: “what can you do for me?
    - An ATM machine object will have the following operations: Withdraw, print receipt, swipe card, and so on.
    - An object pen can have the following operations: Write, leak ink, etc.
- Each object has knowledge (attribute) about its current state:
    - The ATM object has the following attributes: cash on hand, cards recognized, ATM code, etc.
    - The pen object has the attributes: amount of ink remaining, etc.

- Question: How do you decide what operations and attributes are relevant to your object?

    Are you interested in the age or length of the pen?

- Aggregation: Objects may be made of other objects

5. Messages

- Objects can communicate between each other by sending messages.

- The interactions among objects are described in terms of ‘messages’, which are effectively procedure calls executed in the context of the receiving object.

- A message represents a command sent to an object (recipient or receiver) to perform a certain action by invoking one of the methods of the recipient.

- A message consists of the following:

    - Receiving object,
    - The method to be invoked, and
    - Optionally the arguments of the method.

- Example:

```
Panel1.add(button1);

    Recipient: panel1
    Method add
    Arguments: button1
```

6. Classes

- The world has too many objects and we classify these objects into categories or classes
- A class is a template for a group of objects.
- It is a specification of all the operations and attributes for a type of object.
- Classes can also be viewed as cookie-cutter: each class creates cookies (objects) of the same shape (set of operations, attributes).
- Example:
    - Define a pen class as follows:
        - Operations: write: check the amount of the ink and refuse to write if the pen is empty, refill: Add ink to the pen if it is empty.
        - Attributes: ink color, ink remaining,
    - Create pen objects: my pen, you pen, etc.

- A number of different *relationships* can hold among them:

  Fundamental ones are ***inheritance*** and ***aggregation*** that relate classes to each other, and '***instanceOf***' which relates a class to its instances.

7. Classes and Objects

- Classes are static definitions that enable us to understand all the objects of that class. They are only concepts to help us understand the real world. They do not exist at run time.
- Objects are dynamic entities that exist in the real world and are available at run time.
- An object is an instance of a class.
- OO users use the two words class and object interchangeably.

8. Object-oriented characteristics

- Encapsulation:
  - ✓ OOP *encapsulates* data (*attributes*) and methods (*behaviors*) into *objects;* the data and methods of an object are intimately tied together.
  - ✓ Encapsulation also called **information hiding** is the process of making the detailed implementation of an object hidden from its user.
  - ✓ It allows you to ignore the low-level details of how things work internally and enables you to think at a higher level of abstraction.

    - Do you have to know how your TV works to use it?
    - Try to understand how ATM works?
    - You fly on B747 without knowing the details operations of an airplane.

  - ✓ **Bypassing encapsulation is impossible:**
    ATM: bypassing encapsulation is bank robbery.

  - ✓ Users view objects as black boxes: They know what operations may be requested of an object, but don't how the specifics of how the operation are performed.

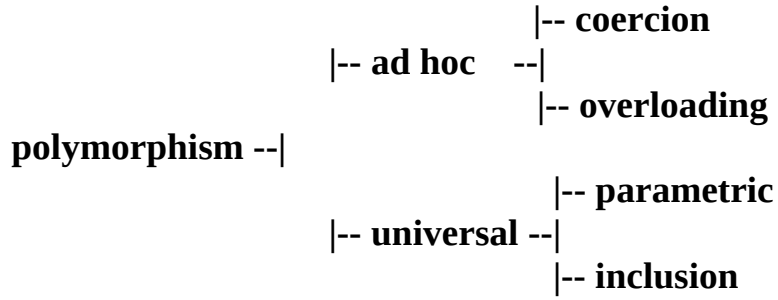
  - ✓ Another advantage of encapsulation is that the implementation of the methods may change without requiring the applications that use them to be modified also as long as the signature of the method does not change.

- **Inheritance**

  - ✓ Inheritance defines a relationship among classes.
  - ✓ Inheritance is also called “is-a-kind-of” relationship: an object of the subclass is a kind of object of the superclass.
  - ✓ Example:
    - A deposit bank account object can be viewed as a kind of a bank account.
  - ✓ Subclasses provide specialized behaviors from the basis of common elements provided by the superclass. Through the use of inheritance, programmers can reuse the code in the superclass many times.

- **Polymorphism:**

  - Polymorphism (Greek for many forms) means that the same operation can be defined for many different classes, and each can be implemented in their different way.
  - Luca Cardelli and Peter Wegner classified polymorphism into the following categories:



  - **Coercion**
    - Coercion represents implicit parameter type conversion to the type expected by a method or an operator, thereby avoiding type errors.
    - Example:

```
double d = 764.5f;
```

      d is declared double and it is holding a value of type float.

- **Overloading**
  - Overloading allows the use of the same operator or method name to denote multiple implementation of a method.
  - Example:

    public void draw (Line firstLine);
    public void draw (Square firstsquare)

    draw can be used polymorphically with Line and Square types.

- **Parametric**
  - Parametric polymorphism allows the use of a single abstraction across many types.
  - Universal parametric:
    - Java 1.5's Generics and C++'s Templates

Without Parameter Polymorphism:

```
// create a collection. just holds “objects”
Vector strV = new Vector();
// add an object
strV.add(new String(“csci210”));
// Retrieving the object requires a cast. Otherwise, there will // be an error.
String s = (String) strV.get(0);
```

With Parameter Polymorphism:

```
// create a collection that holds String type
Vector strV = new Vector<String>();
// add an object
strV.add(new String(“csci210”));
// No need for casting to retrieve the object. (safety)
String s = strV.get(0);
```

- **Inclusion**

  - Inclusion polymorphism achieves polymorphic behavior through an inclusion relation between types or sets of values.
  - For many object-oriented languages the ***inclusion relation*** is a **subtype** relation.
  - Occurs in languages that allow subtypes and inheritance.
  - An instance of an object of a given type can use the functions of its supertype.
  - Example:

```
List<String> text = new ArrayList<String>();
String title = text.get(i).toUppercase();
```

    - text object uses:
      - **get()** from the ArrayList class, and
      - **toUppercase()** method from the String class

- Another example:
  - Strongly-typed OO languages support variables that refer to any variable of a class, or to variables of derived classes:

```
class Stack  {
 public:
   void push( int i )  { elements[top++] = i; };
   int pop( )  { return( elements[--top] ); };
 private:
   int elements[100];
   int top = 0;
};
class CountingStack: public Stack  {
 public:
   int size( ); // Returns the num. of elements
};
```

  - ✓ A **Stack** pointer may also point to a **CountingStack**, but not vice versa:

```
Stack *sp = new Stack;
CountingStack *csp = new CountingStack;
sp = csp; // Okay
csp = sp; // Statically not sure; disallow
```

  - ✓ If we tried to access the **size( )** member function of **csp** after the **csp = sp;** statement, what happens? (Run-time error)

  - ✓ If we do not use pointers, we have similar problems

```
Stack s;
CountingStack cs;
s = cs;  // Okay: Coercion
cs = s;  // Not allowed
```

9. Implementation of Object-Oriented Constructs

- Two important questions for programming language implementers are:
  - Storage structures for instance variables
  - Dynamic bindings of messages to methods

- Instance Data Storage:

  - Class Instance Record (**CIR**): static (defined at compile time) structure similar to the C/C++ struct implementation.
  - Subclasses of a class can extend the CIR of the base class to add new instance variables.

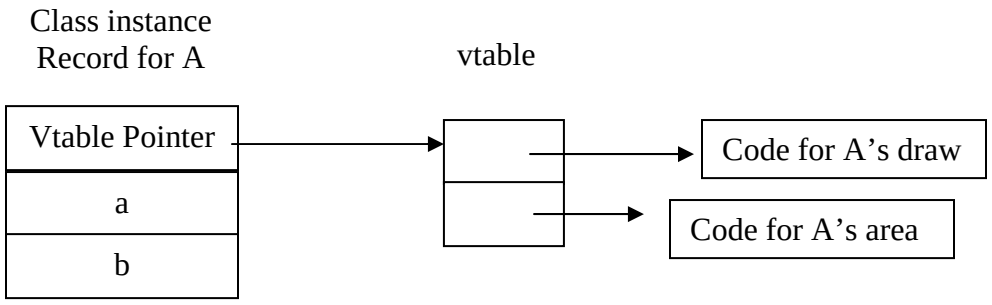
- Dynamic Binding of Methods Calls to Methods

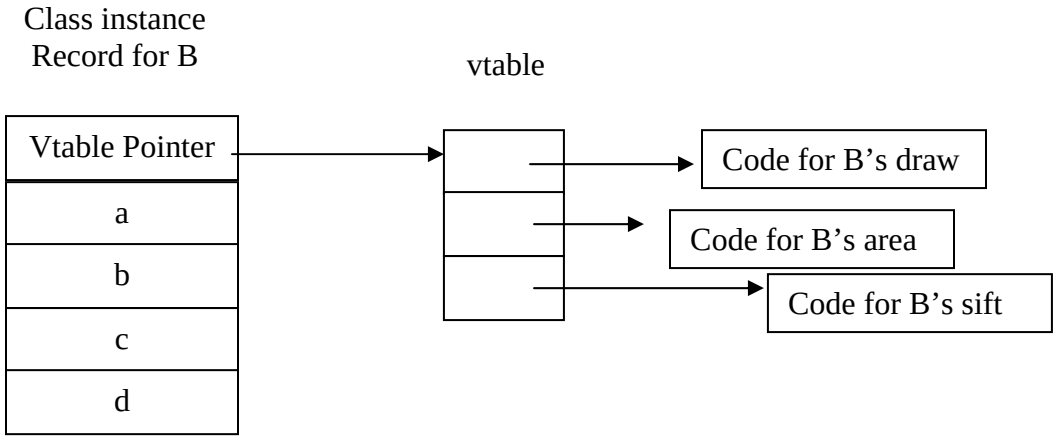
  - Use CIR structure
  - Each object would need a pointer to each method.
  - We need a pointer to the code of the called method:
    - An entry in the CIR structure object for each method.
    - Drawback: a pointer for each method for each object.

- Virtual Method Table: vtable
  - Note that the list of dynamically bound methods that can be called from object is the same for all objects.
  - We only need a single table to store pointers for these methods: vtable.

- Example:

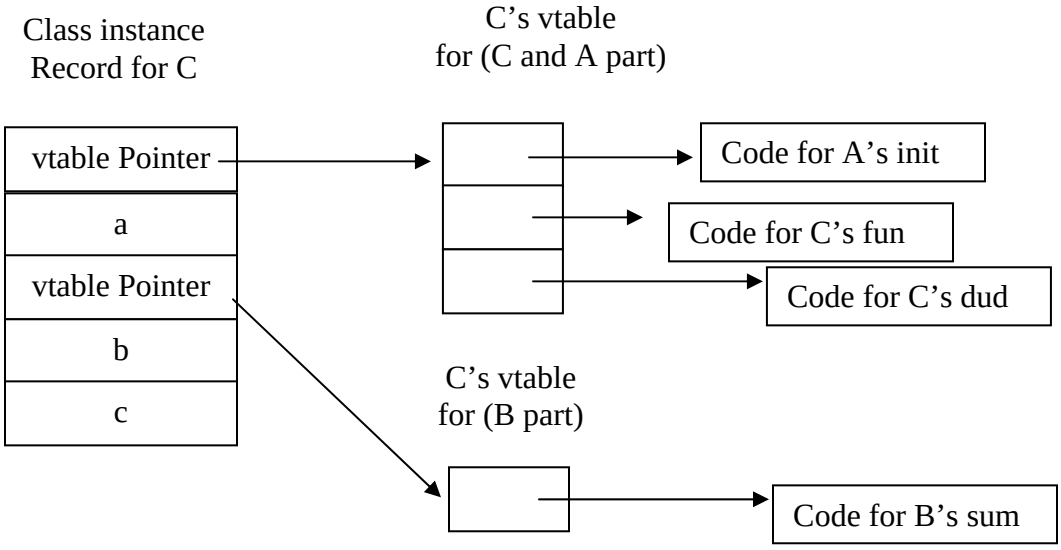
```
public class A{
        public int a, b;
        public void draw() { … }
        public void area() { … }
}
public class B extends A{
        public int c, d;
        public void draw() { … }
        public void area() { … }
        public void sift() { … }
}
```





- Handling Multiple Inheritance: C++
  - It is more complicated than single inheritance implementation.
  - We need a vtable for each parent class.
  - Example:

```
class A {
        public:
                int a;
                virtual void fun(){ … }
                virtual void init() { … }
};
class B{
        public:
                int b
                virtual void sum{ … }
};
class C: public A, public B {
        public:
                int c;
                virtual void fun(){ … }
                virtual void dud() { … }
};
```



10. Dynamic Binding

- Derived classes may also add new private data, and ***redefine*** or ***override*** operations
- If we redefine the **push( )** method of **CountingStack** to count insertions, then we have ***overridden*** the old method:

```
class CountingStack_2: public Stack  {
  public:
    int size( ); // Returns the num. of elements
    void push( int i )  {
                elements[top++] = i;
                ++counter;
    };
  private:
    int counter = 0;
};
```

✓ This introduces some ambiguity as to which method will be called for polymorphic variables:

```
Stack *sp = new Stack;
CountingStack *csp = new CountingStack;
sp->push( 42 ); // Stack::push called
csp->push( 43 ); // CountingStack::push called
sp = csp; // Assignment okay
sp->push( 44 ); // Which push is called?
```

sp->push(44);:
stack::sp->push(); or
counting_stack_2::sp->push();

- **Static binding**:

- Static binding means that the method of the pointer variable type is called: **stack::sp->push();**

- **Dynamic binding**:
  - Dynamic binding means that the method of the object pointed to is called: **counting_stack_2::sp->push();**

11. OO Programming Languages

11.1. C++

- C++ (1985): based upon (and implemented in) C, but with an object oriented emphasis. The goals were to make C++ an equally (or almost so) efficient superset of C, with classes and inheritance added.

- It is a very popular language because of its close ties to C, the ready availability of good compilers, and its effective use of OO.

- Its drawbacks are the same type and error checking weaknesses as C, and perhaps an overabundance of features (many of which are rarely used in practice) which make the language more difficult to learn.

- Use ***constructors*** to create instances **polygon( )**
    - The constructor is called after that of its parent

- Use ***destructor*** to deallocate an object

~polygon( )

    - No garbage collection is provided
    - You have to take care of this in your destructor

- Three levels of member protection are provided
    - All **private** members are only accessible inside the class itself (Same as Java)
    - All **protected** members are accessible inside the class, and all derived classes. (Same as Java)

- All **public** members are generally directly accessible to all clients. (Same as Java)
- All **friend** classes have same rights as the class. (Same as "Default" in Java)

- C++ Virtual functions:
  - C++ virtual function is a member function of a class, whose functionality can be over-ridden in its derived classes.
  - Properties of C++ Virtual Function:

    A member function of a class

    Declared with *virtual* keyword

    Usually has a different functionality in the derived class

    A function call is resolved at run-time

  - The difference between a non-virtual c++ member function and a virtual member function is, the non-virtual member functions are resolved at compile time (Static binding). Where as the c++ virtual member functions are resolved during run-time (Dynamic binding).

  - Example: (http://www.codersource.net/cpp_virtual_functions.html)
    - This article assumes a base class named **Window** with a virtual member function named **Create**. The derived class name will be **CommandButton**, with our over ridden function Create.

```
class Window // Base class for C++ virtual function example
    {
     public:
        // virtual function for C++ virtual function example
        virtual void Create()        {
            cout <<"Base class Window"<<endl;
        }
    };
    class CommandButton : public Window
    {
     public:
        void Create()
        {
           cout<<"Derived class Command Button - Overridden C++
virtual function"<<endl;
        }
    };

    void main()
    {
       Window  *x, *y;

       x = new Window();
       x->Create();

       y = new CommandButton();
       y->Create();
    }
```

- The output of the above program will be,
  Base class Window
  Derived class Command Button
- If the function **had not been declared virtual**, then the base class function would have been called all the times. Because, the function

address would have been statically bound during compile time.

11.2. Java

- **Background**
    - An object-oriented language, formally called ***Oak***, designed by Sun Microsystems Inc. and introduced in 1995.
    - It was designed by James Gasoling for use in consumer electronics.
    - Because of the robustness and platform independent nature of the language, Java moved for the consumer electronics industry into the WWW.
    - An object-oriented language and class of library (Windows management, I/O and network communication, etc.)
    - Use a virtual machine for program execution: Sun's compiler does not compile Java programs to machine language, but to pseudocode executed by a "Java virtual machine" interpreter.

- Key Features of Java:
  - In one of the first papers published by Sun, Java language was described as:

    " A simple, object-oriented, network-savvy, interpreted, robust, secure, architecture neutral, portable, high-performance, multithreaded, dynamic language."

  - Simple:

    - ✓ Java is based on C++, but it omits many rarely used, poorly understood, confusing features of C++
    - ✓ Some of the omitted features primarily consist of explicit dynamic allocation of memory, operator overloading (although the Java language does have method overloading), and extensive automatic coercions.
    - ✓ Another aspect of being simple is being small. One of the goals of Java is to enable the construction of software that can run stand-alone in small machines.
    - ✓ A small size is important for use in embedded systems and so Java can be easily downloaded over the net.

- Object-Oriented:
  - ✓ The object-oriented facilities of Java are essentially those of C
  - ✓ The only unit of programming is the class description: No functions or variables that exist outside the class boundary.

- Network-Savvy
  - ✓ Java was designed with the Internet in mind.
  - ✓ Java has an extensive library of routines for coping easily with TCP/IP protocols like HTTP and FTP. This makes creating network connections much easier than in C or C++.
  - ✓ Java applications can open and access objects across the net via URLs with the same ease that programmers are used to when accessing a local file system.

- Robust:
  - ✓ Java was designed to be graceful in the presence of hardware or software errors: extensive use of exception handlers.
  - ✓ The single biggest difference between Java and C/C++ is that Java has a pointer model that eliminates the possibility of overwriting memory and corrupting data.
  - ✓ Because there are no pointers in Java, programs can't accidentally overwrite the end of a memory buffer. Java programs also cannot gain unauthorized access to memory, which could happen in C or C++.

- Secure
  - ✓ Java is intended for use in networked/distributed environments. Toward that end, a lot of emphasis has been placed on security.
  - ✓ Java enables the construction of virus-free, tamper-free systems.

- There is a strong interplay between "robust" and "secure." For example, the changes to the semantics of pointers make it impossible for applications to forge access to data structures or to access private data in objects that they do not have access to. This closes the door on most activities of viruses.

- Architecture Neutral
  - Java was designed to support applications on networks. In general, networks are composed of a variety of systems with a variety of CPU and operating system architectures.
  - To enable a Java application to execute anywhere on the network, the compiler generates an architecture-neutral object file format--the compiled code is executable on many processors, given the presence of the Java runtime system.
  - Single system software distribution: application writers develop applications that run on all platforms. The Java compiler does this by generating ***bytecode*** instructions that have nothing to do with particular computer architecture.

- Portable
  - Being architecture neutral is a big step toward portability.
  - The exact same system can be compiled on one system, and then executed on many different types of systems.
  - The libraries that are a part of the system define portable interfaces. For example, there is an abstract

Window class and implementations of it for Unix, Windows NT/95, and the Macintosh.

- ✓ The Java system itself is quite portable. The compiler is written in Java

- Interpreted
  - ✓ Java bytecodes are translated on the fly to native machine instructions (interpreted) and not stored anywhere.
  - ✓ Since interpreters are generally much slower in execution, just-in-time (JIT) Compilers were introduced to speedup the execution of java programs.

- High Performance
  - ✓ The bytecodes can be translated on the fly (at runtime) into machine code for the particular CPU the application is running on.
  - ✓ Although initial experiment results showed that the performance of bytecodes converted to machine code was almost indistinguishable from native C or C++, Java program were generally slow in execution.
  - ✓ Now JIT compilers allow platform-independent Java programs to be executed with nearly the same run-time performance as conventional compiled languages.

- Multithreaded

  - ✓ Unfortunately, writing programs that deal with many things happening at once can be much more difficult than writing in the conventional single-threaded C and C++ style.

  - ✓ Java is one of the first languages to be designed explicitly for the possibility of multiple threads of execution running in one program.

  - ✓ Java has a sophisticated set of synchronization primitives that are based on the widely used monitor and condition variable paradigm introduced by C.A.R.Hoare. By integrating these concepts into the language (rather than only in classes) they become much easier to use and are more robust.

1. Dynamic
   - ✓ In a number of ways, Java is a more dynamic language than C or C++. It was designed to adapt to an evolving environment:
     - Binding is done dynamically

11.3. Features Removed from C and C++

No More Typedefs, Defines, or Preprocessor:

- No need for *header files*. Instead of header files, Java language source files provide the declarations of other classes and their methods.
- In Java, you obtain the effects of #define by using constants. You obtain the effects of typedef by declaring classes--after all, a class effectively declares a new type.

2. **No More Structures or Unions:**
   - You can achieve the same effect simply by declaring a class with the appropriate instance variables.

No More Functions:

- Java has no *functions*.
- Anything you can do with a function you can do just as well by defining a class and creating methods for that class

3. No **virtual** keyword in Java. Non-static methods use dynamic binding, so there no need for virtual keyword.

4. **No More Multiple Inheritance:**
   - Multiple inheritance (the ability of a class to inherit from two or more parent classes was discarded from Java.

No More Goto Statements:

- Java has no goto statement.

No More Operator Overloading:

- You cannot overload the standard arithmetic operators.

- Eliminating operator overloading leads to great simplification of code.

5. **No More Automatic Coercions:**
   - Java prohibits *automatic coercions.*
   - You must use casting.

No More Pointers:

- Java has no pointer data types.

11.4. Python

- Classes
  - Syntax:

```
class ClassName:
    <statement-1>
    .
    .
    .
    <statement-N>
```

  - Instance variables in Python are called **Data Attributes**.
  - Unlike C++ and Modula-3, built-in types can be used as base classes for extension by the user.
  - like in C++, most built-in operators with special syntax (arithmetic operators, subscripting etc.) can be redefined for class instances:

    “__add__ “ is used for addition.

- Example:

```
class Fraction:
   "Fraction ADT"
   def __init__(self, __numerator = 0 , __denominator = 1):
      try:
         self.__numerator = __numerator
         if __denominator == 0:
            raise ZeroDivisionError
         self.__denominator = __denominator
      except ZeroDivisionError:
         print "__denominator must a non-zero value. It is set to 1"
         self.__denominator = 1
   # gcd is a private method.
   def __getitem__(self):
         return(self.__numerator)
   def __gcd (self):
      n = self.__numerator
      m = self.__denominator
      if (n == 0 or m == 0):
         return 0
      if (n < 0):
         n = -n
      if (m < 0):
         m = - m
      # Substract the largest from the smallest until they are equal
      while (1):
         if n > m:
            n -= m
         else:
            if n < m:
               m -=n
            else:
               break
      return n
   def reduceF (self):
      self.__numerator = self.__numerator / self.__gcd()
      self.__denominator = self.__denominator / self.__gcd()
   def __add__(f1, f2):
      f = Fraction()
      f.__numerator = f1.__numerator * f2.__denominator + f1.__denominator * f2.__numerator
      f.__denominator = f1.__denominator * f2.__denominator
      f.reduceF()
      return (f)
   def __sub__(f1, f2):
```

```
    f = Fraction()
    f.__numerator = f1.__numerator * f2.__denominator - f1.__denominator * f2.__numerator
    f.__denominator = f1.__denominator + f2.__denominator
    f.reduceF()
    return (f)
  def __mul__(f1, f2):
    f = Fraction()
    f.__numerator = f1.__numerator * f2.__numerator
    f.__denominator = f1.__denominator * f2.__denominator
    f.reduceF()
    return(f)
  def __div__(f1, f2):
    f = Fraction()
    f.__numerator = f1.__numerator * f2.__denominator
    f.__denominator = f1.__denominator * f2.__numerator
    f.reduceF()
    return(f)
  def printF(self):
    print self.__numerator , "/",  self.__denominator
```

Load and test the program:

```
f1 = Fraction(5,6)
f2 = Fraction(2,3)
f3 = Fraction(0,200)
f4 = Fraction(33)
ff = Fraction(0,0)
f5 = Fraction()
fa = Fraction()
fs = Fraction()
fm = Fraction()
fd = Fraction()
print 'f1: ', f1.printF()
print 'f2: ', f2.printF()
print 'f3: ', f3.printF()
print 'f4: ', f4.printF()
print 'ff: ', ff.printF()
print 'f5: ', f5.printF()
fa = f1 + f2
print 'f1 + f2 : ', fa.printF()
fs = f1 - f2
print 'f1 - f2: ', fs.printF()
fm = f1 * f2
print 'f1 * f2: ', fm.printF()
fd = f1 / f2
print 'f1 / f2: ', fd.printF()
```

- Notes:
  - Construction function: Overloading

```
>>> f1 = Fraction(5,6)
f1:  5 / 6
>>> f2 = Fraction(2,3)
f2:  2 / 3
>>> f3 = Fraction(0,200)
f3:  0 / 200
>>> f4 = Fraction(33)
```

```
f4:  33 / 1
>>> f5 = Fraction()
f5:  0 / 1
```

- Operator overloading:

```
def __add__(f1, f2):
    f = Fraction()
    f.__numerator = f1.__numerator * f2.__denominator + f1.__denominator *
f2.__numerator
    f.__denominator = f1.__denominator * f2.__denominator
    f.reduceF()
    return (f)
```

```
>>> fa = f1 + f2
>>> print 'f1 + f2 : ', fa.printF()
f1 + f2 :  3 / 6
```

- Python Modifiers
    - Public
    - Private
    - Protected: It is not supported.

  - Unlike Java pr C++ , Python determine a private attribute by its name:

    - The name of a private attribute starts with **two underscores** and does not end with two underscores
    - All other attributes are public.

    - Try calling a private method:

```
>>> f1.__gcd()

Traceback (most recent call last):
  File "<pyshell#196>", line 1, in
<module>
    f1.__gcd()
AttributeError: Fraction instance has no
attribute '__gcd'
>>>
```

  - Protected:
    - Python has **no concept of protected** class methods.
    - Members are accessed only by their class and inherited classes.

- Instantiation
  - Class instantiation uses function notation:

    x = MyClass()

    Creates a new instance of the class and assigns this object to the local variable x.

  - Invoking Class Methods: Similar to C++/Java

```
>>> f = Fraction(2, 3)
>>> f.printF()
2 / 3
```

  - In addition, Python uses another notation::

    Fraction.printF(f)

```
>>> Fraction.printF(f)
2 / 3
>>>
```

- Inheritance
  - Syntax:

```
class DerivedClassName(BaseClassName):
    <statement-1>
    .
    .
    .
    <statement-N>
```

  - The name BaseClassName must be defined in a scope containing the derived class definition.

  - Multiple Inheritance

    - Python supports multiple inheritance

```
class DerivedClassName(Base1, Base2, Base3):
    <statement-1>
    .
    .
    .
    <statement-N>
```

  - Accessing class attributes Semantic: **Resolution rule**
    - Python uses depth-first, left-to-right: if an attribute is not found in DerivedClassName, it is searched in Base1, then (recursively) in the base classes of Base1, and only if it is not

found there, it is searched in Base2, and so on.

- Overloading:

  - Python has no form of function overloading whatsoever.
  - Methods are defined solely by their name, and there can be only one method per class with a given name.
  - So if a descendant class has a method, it *always* overrides the ancestor method, even if the descendant defines it with a different argument list.
  - Example:

```
class myClass:
    myInstance = 0
    def myMethod(self):
        self.myInstance -= 1
        return(self.myInstance)
    def myMethod(self, inValue):
        self.myInstance += inValue
        return(self.myInstance)

c1 = myClass()
c1.myInstance = 100
print c1.myMethod(2000)
print c1.myMethod()


>>> ================ RESTART ================
>>>
2100

Traceback (most recent call last):
```

```
  File
"C:/bell_documents/abell_docs/courses/python/examples/overload
_one.py", line 13, in <module>
    print c1.myMethod()
TypeError: myMethod() takes exactly 2 arguments (1 given)
>>>
```

Python will only see the last declaration.

- Example:

```
def __init__(self, __numerator = 0 , __denominator = 1):
    try:
        self.__numerator = __numerator
        if __denominator == 0:
            raise ZeroDivisionError
        self.__denominator = __denominator
    except ZeroDivisionError:
            print "__denominator must a non-zero value. It is set to 1"
        self.__denominator = 1



        f1 = Fraction(5,6)
        f2 = Fraction(0,200)
        f3 = Fraction(33)
        f4 = Fraction()
```